RL-TR-96-215
Final Technical Report
February 1997

DYNAMIC BACKTRACKING

University of Oregon

Jointly Sponsored by
Rome Laboratory and
Advanced Research Projects Agency
ARPA Order No. A009

Rome Laboratory
Air Force Materiel Command
Rome, New York

RL-TR-96-215 has been reviewed and is approved for publication.

APPROVED: [illegible]

NORTHRUP FOWLER III
Project Engineer

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

DYNAMIC BACKTRACKING

Contractor:	University of Oregon
Contract Number:	F30602-93-C-0031
Effective Date of Contract:	1 July 1993
Contract Expiration Date:	30 June 1996
Program Code Number:	4E20
Short Title of Work:	Dynamic Backtracking
Period of Work Covered:	Jul 93 - Jun 96

Principal Investigator:	Matthew L. Ginsberg
Phone:	(541) 346-0470
RL Project Engineer:	Northrup Fowler III
Phone:	(315) 330-3011

This research was jointly sponsored by the Defense Advanced Projects Agency of the Department of Defense and by Rome Laboratory, and was monitored by Northrup Fowler III, RL/C3C, 525 Brooks Road, Rome, NY 13441-4505.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE February 1997	3. REPORT TYPE AND DATES COVERED Final Jul 93 - Jun 96

4. TITLE AND SUBTITLE DYNAMIC BACKTRACKING	5. FUNDING NUMBERS C - F30602-93-C-0031 PE - 62301E & 62702F PR - A009 TA - 00 WU - 01
6. AUTHOR(S) Matthew L. Ginsberg, James M. Crawford, and David W. Etherington	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Oregon CIRL, 1269 University of Oregon Eugene, OR 97403-1269	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Research Projects Agency 3701 North Fairfax Drive Arlington, VA 22203-1714 Rome Laboratory/C3C 525 Brooks Road Rome, NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-215

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Northrup Fowler III/C3C/(315) 330-3011

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The goal of this project was to turn the intuitions behind dynamic backtracking into a series of formally verified algorithms, implement the algorithms, and test the results on realistic problems. These goals have been met and exceeded. Dynamic backtracking has been generalized to partial-order dynamic backtracking, and has been formalized, tested on academic benchmarks, and applied (by one of CIRL's industrial partners) to real industrial scheduling problems.

Of equal importance, the search for novel search algorithms for scheduling problems has led beyond dynamic backtracking to include new techniques, such as limited discrepancy search and doubleback optimization, that are currently the best known techniques for benchmark scheduling problems of realistic size and character.

14. SUBJECT TERMS Search, backtracking, scheduling algorithms	15. NUMBER OF PAGES 152
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

1 Introduction **5**

2 Dynamic Backtracking **11**

2.1 Introduction . . . 11
2.2 Preliminaries . . . 12
2.3 Backjumping . . . 15
2.4 Dynamic backtracking . . . 17
2.5 An example . . . 19
2.6 Experimentation . . . 22
2.7 Discussion . . . 25
2.7.1 Why it works . . . 25
2.7.2 Future work . . . 25
2.8 Proofs . . . 29

3 Partial Order Dynamic Backtracking **34**

3.1 Introduction . . . 34
3.2 Constraints and Nogoods . . . 36
3.3 Dynamic Backtracking . . . 40
3.4 Dynamic Backtracking as Local Search . . . 42
3.5 Partial-order Dynamic Backtracking . . . 43
3.6 Arbitrary Movement . . . 46
3.7 Experimental Results . . . 49
3.8 Discussion . . . 51
3.9 Proofs . . . 52

4 The Hazards of Fancy Backtracking **57**

4.1 Introduction . . . 57
4.2 Dynamic backtracking . . . 59
4.3 Experiments . . . 61
4.4 Analysis . . . 62
4.5 Solution . . . 65
4.6 Discussion . . . 66

5 The Application of Satisfiability Algorithms to Scheduling Problems 67
5.1 Introduction . . . 67
5.2 Scheduling . . . 69
5.3 Propositional Satisfiability . . . 70
5.4 Encoding Scheduling Problems as SAT Problems . . . 70
5.5 Three Satisfiability Algorithms . . . 72
5.5.1 Tableau . . . 72
5.5.2 Gsat . . . 72
5.5.3 Isamp . . . 73
5.6 Experimental Results . . . 74
5.7 Discussion . . . 75

6 Limited Discrepancy Search 78
6.1 Introduction . . . 78
6.2 Existing Strategies . . . 79
6.2.1 Iterative Sampling . . . 79
6.2.2 Backtracking . . . 79
6.3 Discrepancies . . . 80
6.4 Comparison with existing methods . . . 81
6.4.1 Wrong Turns . . . 82
6.4.2 Theoretical results . . . 83
6.5 Variations and Extensions . . . 86
6.5.1 Variable Reordering . . . 86
6.5.2 Using Different Heuristics . . . 86
6.5.3 Combining LDS with Bounded Backtrack Search . . . 87
6.5.4 Local Optimization Using LDS . . . 87
6.5.5 Non-Boolean problems . . . 88
6.6 Experimental Results . . . 88
6.7 Discussion . . . 90
6.8 The combined LDS-BBS algorithm . . . 91

7 Scaling Up 92
7.1 Introduction . . . 92
7.2 Resource Constrained Project Scheduling . . . 93
7.3 A Benchmark Problem . . . 93
7.4 Solution Methods . . . 94
7.4.1 Doubleback Optimization . . . 94
7.5 Why the Optimizer Works . . . 96
7.6 Limited Discrepancy Search . . . 97
7.7 Discussion . . . 99

8 Procedural Attachment **100**
8.1 Introduction 100
8.2 Framework 102
8.2.1 Constraint Satisfaction Problems 102
8.2.2 Search Engines 103
8.2.3 Procedures 106
8.2.4 Combining Engines and Procedures 106
8.3 Theory 108
8.4 Discussion 110

9 Broader Issues in Planning **111**

10 Conclusion **117**

References **121**

Abstract

The goal of this project was to turn the intuitions behind dynamic backtracking into a series of formally verified algorithms, implement the algorithms, and test the results on realistic problems. These goals have been met and exceeded. Dynamic backtracking has been generalized to partial-order dynamic backtracking, and has been formalized, tested on academic benchmarks, and applied (by one of CIRL's industrial partners) to real industrial scheduling problems.

Equally importantly, the search for novel search algorithms for scheduling problems has led beyond dynamic backtracking to include new techniques, such as limited discrepancy search and doubleback optimization, that are currently the best known techniques for benchmark scheduling problems of realistic size and character.

Chapter 1

Introduction

Within the planning community, the general attitude toward search has often been, "Search and you're dead." That is, if the available domain knowledge is so weak that search is necessary, then realistically large problems cannot be solved. The attitude within the CSP and scheduling communities has been quite different. The focus there has been on search control: focusing limited computational resources on productive regions within extremely large search spaces. The work reported here is about search control and, more specifically, search control for realistically large applications.

Almost all of the algorithms commonly applied to scheduling and related problems can be put into one of three broad classes: tree search (backtracking) algorithms, local search algorithms, and heuristic approaches that do not involve search. Tree search algorithms have attractive formal properties and work quite well on small to medium-sized problems. Heuristic approaches and local search algorithms are the methods of choice for many practical problems: they are generally straightforward to implement and work well on larger problems. One of the primary contributions of the work funded under this project has been an understanding of the limitations of traditional backtracking algorithms, and more importantly, the development of a series of algorithms that have the desirable formal properties of tree search approaches but scale successfully to larger problems.

To illustrate existing approaches, consider a large scheduling problem. We can view such a problem (like many other search problems) as a set of decisions to be made, subject to a set of constraints. For example, in a TPFDD one has to decide which transport to use for each move requirement. The constraints in a TPFDD are the capacities of the transports, the ALD (Available Load Date) and LAD (Latest Arrival Date) for the move requirements, etc.

In tree search approaches, each decision is viewed as a node in the tree, with one child for each way the decision can be made. The size of the overall tree grows exponentially with the number of decisions (since we end up with a leaf node for every combination of ways all the decisions can be made). A variety of techniques have been developed in an attempt to minimize the size of this search tree (arc consistency and other preprocessing techniques, dependency-directed backtracking, etc.), but for problems of realistic size and complexity the trees are still huge. For example, for a set of scheduling problems relevant to aircraft manufacture currently being studied at CIRL, we estimate that the entire search tree has approximately 10^{300} leaves (without using preprocessing or dependency-directed backtracking techniques). TPFDDs appear to be even larger than this.

The fastest current implementations may examine on the order of a billion leaf nodes in a reasonable period of time (say an hour). This means that on large problems one can examine a vanishingly small percentage of the overall search space. This means that we must be careful that we spend our limited time well.

The primary problem with tree-search techniques is that they end up spending their time badly because of a phenomenon we call the *early mistake problem.* Tree search algorithms work by making decisions sequentially. If a dead-end is reached (a point where there is no way to satisfy the constraints given the current set of decisions), the most recent decision is retracted.[1] Now consider what happens if one of the early decisions is made incorrectly. It will then be necessary to search the entire subtree below this mistake before the mistake can be corrected. Effectively this means that if we are searching a billion nodes out of a tree of size 10^{30}, we will search through all the ways to make the last 20 decisions and never reconsider the first 60 decisions.[2] If any of these first 60 decisions are early mistakes then they will never be corrected. To make matters worse, it turns out that these early decisions are often the *most* likely decisions to be wrong. This is because the heuristics used to make decisions are often more accurate in the presence of more information, and the amount of information available increases as we go down the tree.

One way to overcome the early mistake problem is to use *local search.* Local search algorithms always work with a complete (though not necessarily valid) set of decisions – *e.g.*, a complete schedule. At each point, heuristics are used to pick a decision to reconsider. The other options at the chosen decision point are considered and an option is chosen that maximizes some global measure of schedule quality. This avoids the early mistake problem since any decision can be reconsidered at any point. Local search algorithms are currently the method of choice for some problem classes.

In some realistic problem classes, for example some manufacturing scheduling applications and TPFDDs, the current best fielded method seems to be the use of heuristics only, with no search. Under this approach, each decision is made by applying heuristics, but decisions are never reconsidered. If a constraint is violated, that constraint is relaxed. These algorithms can be seen as a degenerate case of tree search or local search in which the first leaf node visited is taken as the final solution.

Despite their drawbacks, tree search algorithms have one key advantage over other approaches: they are systematic. This means that each possible schedule is considered

[1]Or in more sophisticated algorithms, the most recent decision relevant to the current dead-end is retracted.

[2]This assumes a branching factor of three.

at most once (unlike local search approaches that may "wander" over a given schedule any number.of times), and eventually the optimal schedule is found. One way to view the algorithmic work that has gone on at CIRL under this award is as a search for algorithms that solve the early-mistake problem without giving up systematicity.

One such algorithm is partial-order dynamic backtracking (PDB). This algorithm is described in detail in chapter 3. In essence the idea is to allow almost any decision to be reconsidered at any point, but to keep around enough information about the parts of the search space visited to maintain systematicity. One *could* just modify a local search algorithm to record every schedule visited, and make sure that no schedule is ever visited twice, but the memory use (and, more importantly, the memory management overhead) would be prohibitive. We therefore require that PDB never use more than an amount of memory polynomial in the size of the problem. This requirement forces some restrictions on the moves available (there are some points at which some decisions cannot be reconsidered) but PDB still allows nearly the search flexibility of local search without giving up systematicity.

A second example of a systematic local search algorithm is limited discrepancy search (LDS). LDS can best be seen as a way to efficiently add search on top of heuristic methods that currently do no search. Assume we have a heuristic for a scheduling problem. The heuristic gives a single preferred schedule (found by applying the heuristic to each decision in turn). Now consider the set of all variants of this schedule derivable by ignoring the heuristic at one point. In some sense these schedules are "neighbors" of the preferred schedule (since they are reachable by deviating from the heuristic exactly once). If the heuristic is good but imperfect (which is usually the case with heuristics) one can generally expect one or more of these neighboring schedules to be an improvement over the heuristically preferred schedule. As we increase the number of points at which we deviate from the heuristic we examine schedules successively further from the heuristic and so visit successively broader portions of the search space.

This is the essence of LDS. LDS(0) follows the heuristic at all points. LDS(1) deviates from the heuristic exactly once on any path from the root to a leaf node, LDS(2) deviates twice, and so on. When the number of deviations reaches the number of choices, LDS is guaranteed to find the optimal schedule (in this sense LDS is systematic). As described in chapter 6, LDS has been tested on a set of job shop scheduling problems from the literature. Its performance is significantly better than that of tree-search based techniques and is comparable to that of the best known local search techniques. As described in chapter 7, on realistic problems relevant to aircraft manufacture, a combination of LDS with doubleback optimization yields the best schedules currently known.[3]

Finally, while we have had significant success in improving search, efficient and

[3]Doubleback optimization is a variant of a technique called *schedule packing* developed by Barry Fox [19]. We use the term doubleback optimization because the technique was invented independently at CIRL and because there are technical differences between the CIRL work and Fox's work.

effective search techniques for scheduling and combinatorial optimization do not suffice, in themselves, to solve complex planning problems in general. At least two other factors must be taken into account. Firstly, practical planning requires the ability to focus search effort so as to that ensure that issues are considered according to their importance. Secondly, it is important to develop plans that are general and flexible enough to actually be usable, not mere idealizations. Accordingly, we have also focussed on more fundamental issues in automated reasoning, leading to innovations that begin to address each of these requirements.

In confronting the first requirement, we noted that a planning system's knowledge about the world can be expected to be sprinkled with modalities concerning possibility, the planner's or other agents' knowledge or lack of knowledge, etc. We have shown that these modalities can provide clues as to where the reasoner, faced with limited computational resources, might take expedient short cuts. We developed a theory of how such modal operators, in addition to their well-understood semantic role in declarative systems, also mark points at which these systems can interrupt their reasoning in favor of other, perhaps more important, tasks. We have used this idea to describe an interruptible declarative system that gradually refines its responses to queries. Although initial responses may be in error, a correct answer will be provided if sufficient computational resources are available.

As we mentioned, usable plans need to be flexible and general. Flexibility requires modularity: separate aspects of a problem must be, to the extend possible, planned for separately. This allows plans for particular subgoals – and perhaps even the subgoals themselves – to be replaced or amended without invalidating the rest of a plan. Generality dictates that planners cannot presume total specifications of everything that will be going on, but must rather allow for the fact that other things will happen as the plan is executing. *Approximate planning* is an approach we have developed that addresses these issues, producing plans that are more robust and flexible and more modular. The central idea is that plans are not programs: they do not specify exactly what happens at every instance. Rather, they are recipes: plans specify what must be done, in addition to whatever else is being done for independent reasons, to achieve a goal. An approximate plan is a plan that will "generally" achieve the goal, assuming pathological actions are not interspersed with the required ones. Such plans can be refined (made less approximate), explicitly planning to preclude undesirable courses of action, until a desired degree of confidence is obtained. As a side effect, approximate planning supports mixed-initiative planning, and produces plans that accord better with people's intuitions about planning. We began to develop the framework for approximate planning under this award, and it has become a major focus of our ARPI Phase III award.

Specific accomplishments funded by, or directly resulting from, this award include the following:

- The dynamic backtracking algorithm has been formalized and generalized to partial-order dynamic backtracking.

- Dynamic backtracking has been implemented and tested on academic benchmarks (so called "crystolographic problems").

- Dynamic backtracking is being used at Honeywell (one of CIRL's industrial affiliates) to solve batch manufacturing problems [31].

- A simplified version of dynamic backtracking ("dynamic backtracking light") has been developed that overcomes the difficulties discussed in chapter 4. This version is still being tested, but it appears to be superior to both TABLEAU and GSAT on the scheduling problems discussed in chapter 5. It is also able to solve SAT encodings of planning problems that are referred to in a paper appearing at AAAI-96 [40] as being unsolvable by any systematic methods.

- Limited discrepancy search has been developed and experimentally evaluated.

- *Partition search* – a technique that brings the advantages of dependency-directed backtracking to adversarial search – has been developed and experimentally evaluated.

- A scheduler has been built that utilizes both LDS and doubleback optimization to generate the best known results on benchmark problems of realistic size and character.

- A translator has been developed that translates theories expressed in a finite-sorted first-order syntax to propositional theories, and supports procedural attachment (see chapter 8). This greatly simplifies the process of encoding new problems into propositional logic.

- The basic ideas behind approximate planning have been developed. This led directly to our approximate planning work in phase three of the ARPI.

The body of this report is organized as follows. We begin with the basic dynamic backtracking algorithm, and the generalization to partial-order dynamic backtracking. We discuss some of the technical problems encountered in incorporating value propagation into dynamic backtracking. We then focus on scheduling problems and discuss several search techniques that perform particularly well in this domain. Finally we discuss related work on procedural attachment and more general issues in planning and the control of reasoning.

Beside the work discussed here in detail, the following additional papers were supported by this grant:

- Symmetry breaking predicates for search problems. J. Crawford, M. Ginsberg, E. Luks, and A. Roy. *KR-96*

- A new algorithm for generative planning. M. Ginsberg. *KR-96*

- Do computers need common sense? M. Ginsberg. *KR-96*
- Experimental results on the crossover point in random 3SAT. J. Crawford and L. Auton. *Artificial Intelligence*, vol. 81, 1996
- Implicates and prime implicates in random 3SAT. B. Schrag and J. Crawford *Artificial Intelligence*, vol. 81, 1996
- Toward efficient default reasoning. D. Etherington and J. Crawford. *AAAI-96*
- When is "early commitment" in plan generation a good idea? D. Joslin and M. Pollack. *AAAI-96*
- Path-based rules in object-oriented programming. J. Crawford, D. Dvorak, D. Litman, A. Mishra, and P. Patel-Schneider. *AAAI-96*
- Partition search. M. Ginsberg. *AAAI-96*
- Tuning local search for satisfiability testing. A. Parkes and P. Walser. *AAAI-96*
- Modality and interrupts. M. Ginsberg. *Journal of Automated Reasoning*, 1995.
- Approximate planning. M. Ginsberg. *Artificial Intelligence*, 1995.
- K-best: A new method for real-time decision making. J. Pemberton. *IJCAI-95*
- Device-structured monitoring: A middle ground. J. Crawford, D. Dvorak, D. Litman, A. Mishra, and P. Patel-Schneider. *IJCAI-95*
- Underlying semantics for the assessment of Reiter's solution to the frame problem. T. Bedrax-Weiss. *SBIA-95*
- Easy to be hard: Difficult problems for greedy algorithms. K. Konolige. *KR-94*

Chapter 2

Dynamic Backtracking

We begin with the basic dynamic backtracking algorithm. This material appeared in "Dynamic backtracking" by Ginsberg, in JAIR 1:25-46.

2.1 Introduction

Imagine that you are trying to solve some constraint-satisfaction problem, or CSP. In the interests of definiteness, I will suppose that the CSP in question involves coloring a map of the United States subject to the restriction that adjacent states be colored differently.

Imagine we begin by coloring the states along the Mississippi, thereby splitting the remaining problem in two. We now begin to color the states in the western half of the country, coloring perhaps half a dozen of them before deciding that we are likely to be able to color the rest. Suppose also that the last state colored was Arizona.

At this point, we change our focus to the eastern half of the country. After all, if we can't color the eastern half because of our coloring choices for the states along the Mississippi, there is no point in wasting time completing the coloring of the western states.

We successfully color the eastern states and then return to the west. Unfortunately, we color New Mexico and Utah and then get stuck, unable to color (say) Nevada. What's more, backtracking doesn't help, at least in the sense that changing the colors for New Mexico and Utah alone does not allow us to proceed farther. Depth-first search would now have us backtrack to the eastern states, trying a new color for (say) New York in the vain hope that this would solve our problems out West.

This is obviously pointless; the blockade along the Mississippi makes it impossible for New York to have any impact on our attempt to color Nevada or other western states. What's more, we are likely to examine every *possible* coloring of the eastern states before addressing the problem that is actually the source of our difficulties.

The solutions that have been proposed to this involve finding ways to backtrack directly to some state that might actually allow us to make progress, in this case Arizona or earlier. Dependency-directed backtracking [70] involves a direct backtrack to the source of the difficulty; backjumping [21] avoids the computational overhead of this technique by using syntactic methods to estimate the point to which backtrack is necessary.

In both cases, however, note that although we backtrack to the source of the problem, we backtrack *over* our successful solution to half of the original problem, discarding our solution to the problem of coloring the states in the East. And once again, the problem is worse than this – after we recolor Arizona, we are in danger of solving the East yet again before realizing that our new choice for Arizona needs to be changed after all. We won't examine every possible coloring of the eastern states, but we are in danger of rediscovering our successful coloring an exponential number of times.

This hardly seems sensible; a human problem solver working on this problem would simply ignore the East if possible, returning directly to Arizona and proceeding. Only if the states along the Mississippi needed new colors would the East be reconsidered – and even then only if no new coloring could be found for the Mississippi that was consistent with the eastern solution.

In this paper we formalize this technique, presenting a modification to conventional search techniques that is capable of backtracking not only to the most recently expanded node, but also directly to a node elsewhere in the search tree. Because of the dynamic way in which the search is structured, we refer to this technique as *dynamic backtracking.*

A more specific outline is as follows: We begin in the next section by introducing a variety of notational conventions that allow us to cast both existing work and our new ideas in a uniform computational setting. Section 2.3 discusses backjumping, an intermediate between simple chronological backtracking and our ideas, which are themselves presented in Section 2.4. An example of the dynamic backtracking algorithm in use appears in Section 2.5 and an experimental analysis of the technique in Section 2.6. A summary of our results and suggestions for future work are in Section 2.7. All proofs have been deferred to the last section in the interests of continuity of exposition.

2.2 Preliminaries

Definition 2.2.1 *By a* constraint satisfaction problem (I, V, κ) *we will mean a set* I *of variables; for each* $i \in I$*, there is a set* V_i *of possible values for the variable* i*.* κ *is a set of constraints, each a pair* (J, P) *where* $J = (j_1, \ldots, j_k)$ *is an ordered subset of* I *and* P *is a subset of* $V_{j_1} \times \cdots \times V_{j_k}$*.*

A solution *to the* CSP *is a set* v_i *of values for each of the variables in* I *such that* $v_i \in V_i$ *for each* i *and for every constraint* (J, P) *of the above form in* κ*,* $(v_{j_1}, \ldots, v_{j_k}) \in P$*.*

In the example of the introduction, I is the set of states and V_i is the set of possible colors for the state i. For each constraint, the first part of the constraint is a pair of adjacent states and the second part is a set of allowable color combinations for these states.

Our basic plan in this paper is to present formal versions of the search algorithms described in the introduction, beginning with simple depth-first search and proceeding to backjumping and dynamic backtracking. As a start, we make the following definition of a partial solution to a CSP:

Definition 2.2.2 *Let (I, V, κ) be a* CSP. *By a* partial solution *to the* CSP *we mean an ordered subset $J \subseteq I$ and an assignment of a value to each variable in J.*

We will denote a partial solution by a tuple of ordered pairs, where each ordered pair (i, v) assigns the value v to the variable i. For a partial solution P, we will denote by $\overline{P}$ the set of variables assigned values by P.

Constraint-satisfaction problems are solved in practice by taking partial solutions and extending them by assigning values to new variables. In general, of course, not any value can be assigned to a variable because some are inconsistent with the constraints. We therefore make the following definition:

Definition 2.2.3 *Given a partial solution P to a* CSP, *an* eliminating explanation *for a variable i is a pair (v, S) where $v \in V_i$ and $S \subseteq \overline{P}$. The intended meaning is that i cannot take the value v because of the values already assigned by P to the variables in S. An* elimination mechanism *ε for a* CSP *is a function that accepts as arguments a partial solution P, and a variable $i \notin \overline{P}$. The function returns a (possibly empty) set $\varepsilon(P, i)$ of eliminating explanations for i.*

For a set E of eliminating explanations, we will denote by $\widehat{E}$ the values that have been identified as eliminated, ignoring the reasons given. We therefore denote by $\widehat{\varepsilon}(P, i)$ the set of values eliminated by elements of $\varepsilon(P, i)$.

Note that the above definition is somewhat flexible with regard to the amount of work done by the elimination mechanism – all values that violate completed constraints might be eliminated, or some amount of lookahead might be done. We will, however, make the following assumptions about all elimination mechanisms:

1. They are *correct.* For a partial solution P, if the value $v_i \notin \widehat{\varepsilon}(P, i)$, then every constraint (S, T) in κ with $S \subseteq \overline{P} \cup \{i\}$ is satisfied by the values in the partial solution and the value v_i for i. These are the constraints that are complete after the value v_i is assigned to i.

2. They are *complete.* Suppose that P is a partial solution to a CSP, and there is some solution that extends P while assigning the value v to i. If P' is an extension of P with $(v, E) \in \varepsilon(P', i)$, then

$$E \cap (\overline{P'} - \overline{P}) \neq \emptyset \tag{2.1}$$

In other words, whenever P can be successfully extended after assigning v to i but P' cannot be, at least one element of $P' - P$ is identified as a possible reason for the problem.

3. They are *concise.* For a partial solution P, variable i and eliminated value v, there is at most a single element of the form $(v, E) \in \varepsilon(P, i)$. Only one reason is given why the variable i cannot have the value v.

Lemma 2.2.4 *Let ε be a complete elimination mechanism for a* CSP, *let P be a partial solution to this* CSP *and let $i \notin \overline{P}$. Now if P can be successfully extended to a complete solution after assigning i the value v, then $v \notin \widehat{\varepsilon}(P, i)$.*

I apologize for the swarm of definitions, but they allow us to give a clean description of depth-first search:

Algorithm 2.2.5 (Depth-first search) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ε:*

1. *Set $P = \emptyset$. P is a partial solution to the* CSP. *Set $E_i = \emptyset$ for each $i \in I$; E_i is the set of values that have been eliminated for the variable i.*

2. *If $\overline{P} = I$, so that P assigns a value to every element in I, it is a solution to the original problem. Return it. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = \widehat{\varepsilon}(P, i)$, the values that have been eliminated as possible choices for i.*

3. *Set $S = V_i - E_i$, the set of remaining possibilities for i. If S is nonempty, choose an element $v \in S$. Add (i, v) to P, thereby setting i's value to v, and return to step 2.*

4. *If S is empty, let (j, v_j) be the last entry in P; if there is no such entry, return failure. Remove (j, v_j) from P, add v_j to E_j, set $i = j$ and return to step 3.*

We have written the algorithm so that it returns a single answer to the CSP; the modification to accumulate all such answers is straightforward.

The problem with Algorithm 2.2.5 is that it looks very little like conventional depth-first search, since instead of recording the unexpanded children of any particular node, we are keeping track of the *failed siblings* of that node. But we have the following:

Lemma 2.2.6 *At any point in the execution of Algorithm 2.2.5, if the last element of the partial solution P assigns a value to the variable i, then the unexplored siblings of the current node are those that assign to i the values in $V_i - E_i$.*

Proposition 2.2.7 *Algorithm 2.2.5 is equivalent to depth-first search and therefore complete.*

As we have remarked, the basic difference between Algorithm 2.2.5 and a more conventional description of depth-first search is the inclusion of the elimination sets E_i. The conventional description expects nodes to include pointers back to their parents; the siblings of a given node are found by examining the children of that node's parent. Since we will be reorganizing the space as we search, this is impractical in our framework.

It might seem that a more natural solution to this difficulty would be to record not the values that have been *eliminated* for a variable i, but those that remain to be considered. The technical reason that we have not done this is that it is much easier to maintain elimination information as the search progresses. To understand this at an intuitive level, note that when the search backtracks, the conclusion that has implicitly been drawn is that a particular node fails to expand to a solution, as opposed to a conclusion about the currently unexplored portion of the search space. It should be little surprise that the most efficient way to manipulate this information is by recording it in approximately this form.

2.3 Backjumping

How are we to describe dependency-directed backtracking or backjumping in this setting? In these cases, we have a partial solution and have been forced to backtrack; these more sophisticated backtracking mechanisms use information about the *reason* for the failure to identify backtrack points that might allow the problem to be addressed. As a start, we need to modify Algorithm 2.2.5 to maintain the explanations for the eliminated values:

Algorithm 2.3.1 *Given as inputs a constraint-satisfaction problem and an elimination mechanism ε:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$. E_i is a set of eliminating explanations for i.*
2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = \varepsilon(P, i)$.*
3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*
4. *If S is empty, let (j, v_j) be the last entry in P; if there is no such entry, return failure. Remove (j, v_j) from P. We must have $\widehat{E}_i = V_i$, so that every value for i has been eliminated; let E be the set of all variables appearing in the explanations for each eliminated value. Add $(v_j, E - \{j\})$ to E_j, set $i = j$ and return to step 3.*

Lemma 2.3.2 *Let P be a partial solution obtained during the execution of Algorithm 2.3.1, and let $i \in \overline{P}$ be a variable assigned a value by P. Now if $P' \subseteq P$ can be* *successfully extended to a complete solution after assigning i the value v but $(v, E) \in E_i$, we must have*

$$E \cap (\overline{P} - \overline{P'}) \neq \emptyset$$

In other words, the assignment of a value to some variable in $\overline{P} - \overline{P'}$ is correctly identified as the source of the problem.

Note that in step 4 of the algorithm, we could have added $(v_j, E \cap \overline{P})$ instead of $(v_j, E - \{j\})$ to E_j; either way, the idea is to remove from E any variables that are no longer assigned values by P.

In backjumping, we now simply change our backtrack method; instead of removing a single entry from P and returning to the variable assigned a value prior to the problematic variable i, we return to a variable that has actually had an impact on i. In other words, we return to some variable in the set E.

Algorithm 2.3.3 (Backjumping) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ε:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$.*
2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = \varepsilon(P, i)$.*
3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*
4. *If S is empty, we must have $\widehat{E}_i = V_i$. Let E be the set of all variables appearing in the explanations for each eliminated value.*
5. *If $E = \emptyset$, return failure. Otherwise, let (j, v_j) be the last entry in P such that $j \in E$. Remove from P this entry and any entry following it. Add $(v_j, E \cap \overline{P})$ to E_j, set $i = j$ and return to step 3.*

In step 5, we add $(v_j, E \cap \overline{P})$ to E_j, removing from E any variables that are no longer assigned values by P.

Proposition 2.3.4 *Backjumping is complete and always expands fewer nodes than does depth-first search.*

Let us have a look at this in our map-coloring example. If we have a partial coloring P and are looking at a specific state i, suppose that we denote by C the set of colors that are obviously illegal for i because they conflict with a color already assigned to one of i's neighbors.

One possible elimination mechanism returns as $\varepsilon(P, i)$ a list of $(c, \overline{P})$ for each color $c \in C$ that has been used to color a neighbor of i. This reproduces depth-first search, since we gradually try all possible colors but have no idea what went wrong when we need to backtrack since every colored state is included in $\overline{P}$. A far more sensible choice would take $\varepsilon(P, i)$ to be a list of $(c, \{n\})$ where n is a neighbor that is already colored c. This would ensure that we backjump to a neighbor of i if no coloring for i can be found.

If this causes us to backjump to another state j, we will add i's neighbors to the eliminating explanation for j's original color, so that if we need to backtrack still further, we consider neighbors of either *i or j*. This is as it should be, since changing the color of one of i's other neighbors might allow us to solve the coloring problem by reverting to our original choice of color for the state j.

We also have:

Proposition 2.3.5 *The amount of space needed by backjumping is $o(i^2 v)$, where $i = |I|$ is the number of variables in the problem and v is the number of values for that variable with the largest value set V_i.*

This result contrasts sharply with an approach to CSPs that relies on truth-maintenance techniques to maintain a list of nogoods [17]. There, the number of nogoods found can grow linearly with the time taken for the analysis, and this will typically be exponential in the size of the problem. Backjumping avoids this problem by resetting the set E_i of eliminating explanations in step 2 of Algorithm 2.3.3.

The description that we have given is quite similar to that developed in [4]. The explanations there are somewhat coarser than ours, listing all of the variables that have been involved in *any* eliminating explanation for a particular variable in the CSP, but the idea is essentially the same. Bruynooghe's eliminating explanations can be stored in $o(i^2)$ space (instead of $o(i^2 v)$), but the associated loss of information makes the technique less effective in practice. This earlier work is also a description of backjumping only, since intermediate information is erased as the search proceeds.

2.4 Dynamic backtracking

We finally turn to new results. The basic problem with Algorithm 2.3.3 is not that it backjumps to the wrong place, but that it needlessly erases a great deal of the work that has been done thus far. At the very least, we can retain the values selected for variables that are backjumped over, in some sense moving the backjump variable to the end of the partial solution in order to replace its value without modifying the values of the variables that followed it.

There is an additional modification that will probably be clearest if we return to the example of the introduction. Suppose that in this example, we color only *some* of the eastern states before returning to the western half of the country. We reorder the variables in order to backtrack to Arizona and eventually succeed in coloring the West without disturbing the colors used in the East.

Unfortunately, when we return East backtracking is required and we find ourselves needing to change the coloring on some of the eastern states with which we dealt earlier. The ideas that we have presented will allow us to avoid erasing our solution to the problems out West, but if the search through the eastern states is to be efficient, we will need to retain the information we have about the portion of the East's search space that has been eliminated. After all, if we have determined that New York cannot be colored yellow, our changes in the West will not reverse this conclusion – the Mississippi really does isolate one section of the country from the other.

The machinery needed to capture this sort of reasoning is already in place. When we backjump over a variable k, we should retain not only the choice of value for k, but also k's elimination set. We do, however, need to remove from this elimination set any entry that involves the eventual backtrack variable j, since these entries are no longer valid – they depend on the assumption that j takes its old value, and this assumption is now false.

Algorithm 2.4.1 (Dynamic backtracking I) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ε:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$.*

2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = E_i \cup \varepsilon(P, i)$.*

3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*

4. *If S is empty, we must have $\widehat{E}_i = V_i$; let E be the set of all variables appearing in the explanations for each eliminated value.*

5. *If $E = \emptyset$, return failure. Otherwise, let (j, v_j) be the last entry in P such that $j \in E$. Remove (j, v_j) from P and, for each variable k assigned a value after j, remove from E_k any eliminating explanation that involves j. Set*

$$E_j = E_j \cup \varepsilon(P, j) \cup \{(v_j, E \cap \overline{P})\} \tag{2.2}$$

   so that v_j is eliminated as a value for j because of the values taken by variables in $E \cap \overline{P}$. The inclusion of the term $\varepsilon(P, j)$ incorporates new information from variables that have been assigned values since the original assignment of v_j to j. Now set $i = j$ and return to step 3.

Theorem 2.4.2 *Dynamic backtracking always terminates and is complete. It continues to satisfy Proposition 2.3.5 and can be expected to expand fewer nodes than backjumping provided that the goal nodes are distributed randomly in the search space.*

The essential difference between dynamic and dependency-directed backtracking is that the structure of our eliminating explanations means that we only save nogood information based on the current values of assigned variables; if a nogood depends on outdated information, we drop it. By doing this, we avoid the need to retain an exponential amount of nogood information. What makes this technique valuable is that (as stated in the theorem) termination is still guaranteed.

There is one trivial modification that we can make to Algorithm 2.4.1 that is quite useful in practice. After removing the current value for the backtrack variable j, Algorithm 2.4.1 immediately replaces it with another. But there is no real reason to do this; we could instead pick a value for an entirely different variable:

Algorithm 2.4.3 (Dynamic backtracking) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ε:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$.*

2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = E_i \cup \varepsilon(P, i)$.*

3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*

4. *If S is empty, we must have $\widehat{E}_i = V_i$; let E be the set of all variables appearing in the explanations for each eliminated value.*

5. *If $E = \emptyset$, return failure. Otherwise, let (j, v_j) be the last entry in P that binds a variable appearing in E. Remove (j, v_j) from P and, for each variable k assigned a value after j, remove from E_k any eliminating explanation that involves j. Add $(v_j, E \cap \overline{P})$ to E_j and return to step 2.*

2.5 An example

In order to make Algorithm 2.4.3 a bit clearer, suppose that we consider a small map-coloring problem in detail. The map is shown in Figure 2.1 and consists of five countries: Albania, Bulgaria, Czechoslovakia, Denmark and England. We will assume (wrongly!) that the countries border each other as shown in the figure, where countries are denoted by nodes and border one another if and only if there is an arc connecting them.

In coloring the map, we can use the three colors red, yellow and blue. We will typically abbreviate the country names to single letters in the obvious way.

We begin our search with Albania, deciding (say) to color it red. When we now look at Bulgaria, no colors are eliminated because Albania and Bulgaria do not share a border; we decide to color Bulgaria yellow. (This is a mistake.)

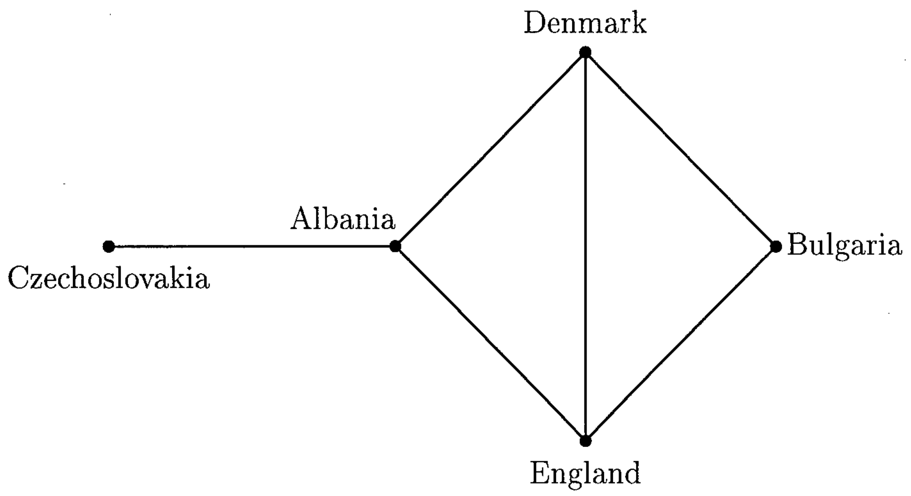

Figure 2.1: A small map-coloring problem

We now go on to consider Czechoslovakia; since it borders Albania, the color red is eliminated. We decide to color Czechoslovakia blue and the situation is now this:

country	color	red	yellow	blue
Albania	red			
Bulgaria	yellow			
Czechoslovakia	blue	A		
Denmark				
England				

For each country, we indicate its current color and the eliminating explanations that mean it cannot be colored each of the three colors (when such explanations exist). We now look at Denmark.

Denmark cannot be colored red because of its border with Albania and cannot be colored yellow because of its border with Bulgaria; it must therefore be colored blue. But now England cannot be colored any color at all because of its borders with Albania, Bulgaria and Denmark, and we therefore need to backtrack to one of these three countries. At this point, the elimination lists are as follows:

country	color	red	yellow	blue
Albania	red			
Bulgaria	yellow			
Czechoslovakia	blue	A		
Denmark	blue	A	B	
England		A	B	D

We backtrack to Denmark because it is the most recent of the three possibilities, and begin by removing any eliminating explanation involving Denmark from the above table to get:

country	color	red	yellow	blue
Albania	red			
Bulgaria	yellow			
Czechoslovakia	blue	A		
Denmark		A	B	
England		A	B	

Next, we add to Denmark's elimination list the pair

$$(\text{blue}, \{A, B\})$$

This indicates correctly that because of the current colors for Albania and Bulgaria, Denmark cannot be colored blue (because of the subsequent dead end at England). Since every color is now eliminated, we must backtrack to a country in the set $\{A, B\}$. Changing Czechoslovakia's color won't help and we must deal with Bulgaria instead. The elimination lists are now:

country	color	red	yellow	blue
Albania	red			
Bulgaria				
Czechoslovakia	blue	A		
Denmark		A	B	A,B
England		A	B	

We remove the eliminating explanations involving Bulgaria and also add to Bulgaria's elimination list the pair

$$(\text{yellow}, A)$$

indicating correctly that Bulgaria cannot be colored yellow because of the current choice of color for Albania (red).

The situation is now:

country	color	red	yellow	blue
Albania	red			
Czechoslovakia	blue	A		
Bulgaria			A	
Denmark		A		
England		A		

We have moved Bulgaria past Czechoslovakia to reflect the search reordering in the algorithm. We can now complete the problem by coloring Bulgaria red, Denmark either yellow or blue, and England the color not used for Denmark.

This example is almost trivially simple, of course; the thing to note is that when we changed the color for Bulgaria, we retained both the blue color for Czechoslovakia and the information indicating that none of Czechoslovakia, Denmark and England could be red. In more complex examples, this information may be very hard-won and retaining it may save us a great deal of subsequent search effort.

Another feature of this specific example (and of the example of the introduction as well) is that the computational benefits of dynamic backtracking are a consequence of the automatic realization that the problem splits into disjoint subproblems. Other authors have also discussed the idea of applying divide-and-conquer techniques to CSPs [58, 79], but their methods suffer from the disadvantage that they constrain the order in which unassigned variables are assigned values, perhaps at odds with the common heuristic of assigning values first to those variables that are most tightly constrained. Dynamic backtracking can also be expected to be of use in situations where the problem in question does *not* split into two or more disjoint subproblems.[1]

2.6 Experimentation

Dynamic backtracking has been incorporated into the crossword-puzzle generation program described in [28], and leads to significant performance improvements in that restricted domain. More specifically, the method was tested on the problem of generating 19 puzzles of sizes ranging from 2×2 to 13×13; each puzzle was attempted 100 times using both dynamic backtracking and simple backjumping. The dictionary was shuffled between solution attempts and a maximum of 1000 backtracks were permitted before the program was deemed to have failed.

In both cases, the algorithms were extended to include iterative broadening [29], the cheapest-first heuristic and forward checking. Cheapest-first has also been called "most constrained first" and selects for instantiation that variable with the fewest number of remaining possibilities (i.e., that variable for which it is cheapest to enumerate the possible values [66]). Forward checking prunes the set of possibilities for crossing words whenever a new word is entered and constitutes our experimental choice of elimination mechanism: at any point, words for which there is no legal crossing word are eliminated. This ensures that no word will be entered into the crossword if the word has *no* potential crossing words at some point. The cheapest-first heuristic would identify the problem at the next step in the search, but forward checking reduces the number of backtracks substantially. The "least-constraining" heuristic [28] was *not* used; this heuristic suggests that each word slot be filled with the word that minimally constrains the subsequent search. The heuristic was not used because it would invalidate the technique of shuffling the dictionary between solution attempts in order to gather useful statistics.

The table in Figure 2.2 indicates the number of successful solution attempts (out of 100) for each of the two methods on each of the 19 crossword frames. Dynamic backtracking is more successful in six cases and less successful in none.

With regard to the number of nodes expanded by the two methods, consider the

[1]I am indebted to David McAllester for these observations.

Frame	Dynamic backtracking	Backjumping	Frame	Dynamic backtracking	Backjumping
1	100	100	11	100	98
2	100	100	12	100	100
3	100	100	13	100	100
4	100	100	14	100	100
5	100	100	15	99	14
6	100	100	16	100	26
7	100	100	17	100	30
8	100	100	18	61	0
9	100	100	19	10	0
10	100	100			

Figure 2.2: Number of problems solved successfully

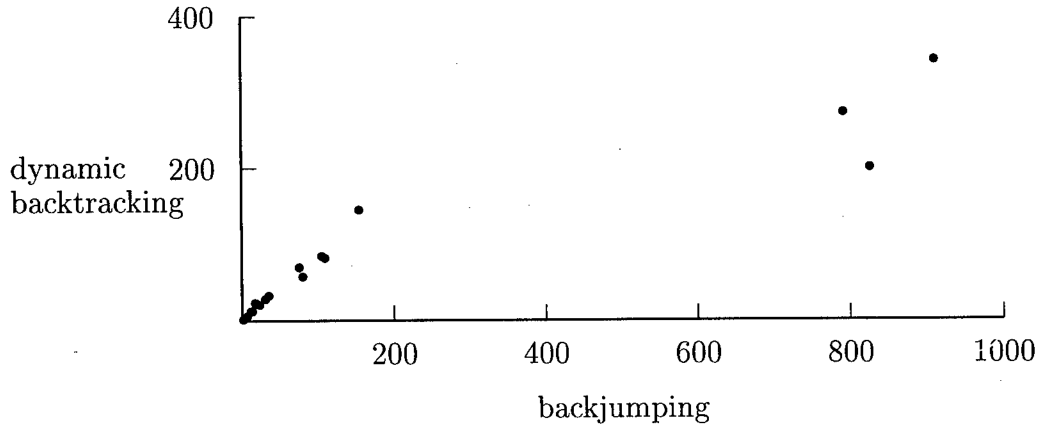

Figure 2.3: Number of backtracks needed

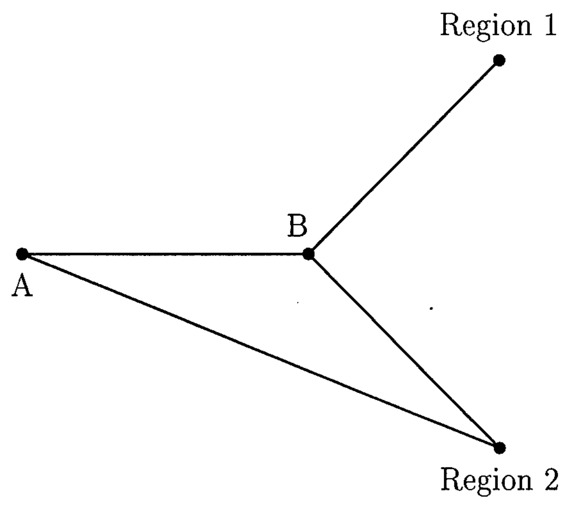

Figure 2.4: A difficult problem for dynamic backtracking

data presented in Figure 2.3, where we graph the average number of backtracks needed by the two methods.[2] Although initially comparable, dynamic backtracking provides increasing computational savings as the problems become more difficult. A somewhat broader set of experiments is described in [39] and leads to similar conclusions.

There are some examples in [39] where dynamic backtracking leads to performance degradation, however; a typical case appears in Figure 2.4.[3] In this figure, we first color A, then B, then the countries in region 1, and then get stuck in region 2.

We now presumably backtrack directly to B, leaving the coloring of region 1 alone. But this may well be a mistake – the colors in region 1 will restrict our choices for B, perhaps making the subproblem consisting of A, B and region 2 more difficult than it might be. If region 1 were easy to color, we would have been better off erasing it even though we didn't need to.

This analysis suggests that dependency-directed backtracking should also fare worse on those coloring problems where dynamic backtracking has trouble, and we are currently extending the experiments of [39] to confirm this. If this conjecture is borne out, a variety of solutions come to mind. We might, for example, record how many backtracks are made to a node such as B in the above figure, and then use this to determine that flexibility at B is more important than retaining the choices made in region 1. The difficulty of finding a coloring for region 1 can also be determined from the number of backtracks involved in the search.

[2]Only 17 points are shown because no point is plotted where backjumping was unable to solve the problem.

[3]The worst performance degradation observed was a factor of approximately 4.

2.7 Discussion

2.7.1 Why it works

There are two separate ideas that we have exploited in the development of Algorithm 2.4.3 and the others leading up to it. The first, and easily the most important, is the notion that it is possible to modify variable order on the fly in a way that allows us to retain the results of earlier work when backtracking to a variable that was assigned a value early in the search.

This reordering should not be confused with the work of authors who have suggested a dynamic choice among the variables that *remain* to be assigned values [18, 28, 55, 81]; we are instead reordering the variables that have *been* assigned values in the search thus far.

Another way to look at this idea is that we have found a way to "erase" the value given to a variable directly as opposed to backtracking to it. This idea has also been explored by Minton *et.al.* in [50] and by Selman *et.al.* in [63]; these authors also directly replace values assigned to variables in satisfiability problems. Unfortunately, the heuristic repair method used is incomplete because no dependency information is retained from one state of the problem solver to the next.

There is a third way to view this as well. The space that we are examining is really a graph, as opposed to a tree; we reach the same point by coloring Albania blue and then Bulgaria red as if we color them in the opposite order. When we decide to backjump from a particular node in the search space, we know that we need to back up until some particular property of that node ceases to hold – and the key idea is that by backtracking along a path *other than* the one by which the node was generated, we may be able to backtrack only slightly when we would otherwise need to retreat a great deal. This observation is interesting because it may well apply to problems other than CSPs. Unfortunately, it is not clear how to guarantee completeness for a search that discovers a node using one path and backtracks using another.

The other idea is less novel. As we have already remarked, our use of eliminating explanations is quite similar to the use of nogoods in the ATMS community; the principal difference is that we attach the explanations to the variables they impact and drop them when they cease to be relevant. (They might become relevant again later, of course.) This avoids the prohibitive space requirements of systems that permanently cache the results of their nogood calculations; this observation also may be extensible beyond the domain of CSPs specifically. Again, there are other ways to view this – Gashnig's notion of *backmarking* [21] records similar information about the reason that particular portions of a search space are known not to contain solutions.

2.7.2 Future work

There are a variety of ways in which the techniques we have presented can be extended; in this section, we sketch a few of the more obvious ones.

Backtracking to older culprits

One extension to our work involves lifting the restriction in Algorithm 2.4.3 that the variable erased always be the most recently assigned member of the set E.

In general, we cannot do this while retaining the completeness of the search. Consider the following example:

Imagine that our CSP involves three variables, x, y and z, that can each take the value 0 or 1. Further, suppose that this CSP has no solutions, in that after we pick any two values for x and for y, we realize that there is no suitable choice for z.

We begin by taking $x = y = 0$; when we realize the need to backtrack, we introduce the nogood

$$x = 0 \supset y \neq 0 \tag{2.3}$$

and replace the value for y with $y = 1$.

This fails, too, but now suppose that we were to decide to backtrack to x, introducing the new nogood

$$y = 1 \supset x \neq 0 \tag{2.4}$$

We change x's value to 1 and erase (2.3).

This also fails. We decide that y is the problem and change its value to 0, introducing the nogood

$$x = 1 \supset y \neq 1$$

but erasing (2.4). And when *this* fails, we are in danger of returning to $x = y = 0$, which we eliminated at the beginning of the example. This loop may cause a modified version of the dynamic backtracking algorithm to fail to terminate.

In terms of the proof of Theorem 2.4.2, the nogoods discovered already include information about all assigned variables, so there is no difference between (2.7) and (2.8). When we drop (2.3) in favor of (2.4), we are no longer in a position to recover (2.3).

We can deal with this by placing conditions on the variables to which we choose to backtrack; the conditions need to be defined so that the proof of Theorem 2.4.2 continues to hold.[4] Experimentation indicates that loops of the form we have described are extremely rare in practice; it may also be possible to detect them directly and thereby retain more substantial freedom in the choice of backtrack point.

This freedom of backtrack raises an important question that has not yet been addressed in the literature: When backtracking to avoid a difficulty of some sort, to where should one backtrack?

Previous work has been constrained to backtrack no further than the most recent choice that might impact the problem in question; any other decision would be both incomplete and inefficient. Although an extension of Algorithm 2.4.3 need not operate under this restriction, we have given no indication of how the backtrack point should be selected.

[4]Another solution appears in [49].

There are several easily identified factors that can be expected to bear on this choice. The first is that there remains a reason to expect backtracking to chronologically recent choices to be the most effective – these choices can be expected to have contributed to the fewest eliminating explanations, and there is obvious advantage to retaining as many eliminating explanations as possible from one point in the search to the next. It is possible, however, to simply identify that backtrack point that affects the fewest number of eliminating explanations and to use that.

Alternatively, it might be important to backtrack to the choice point for which there will be as many new choices as possible; as an extreme example, if there is a variable i for which every value other than its current one has already been eliminated for other reasons, backtracking to i is guaranteed to generate another backtrack immediately and should probably be avoided if possible.

Finally, there is some measure of the "directness" with which a variable bears on a problem. If we are unable to find a value for a particular variable i, it is probably sensible to backtrack to a second variable that shares a constraint with i itself, as opposed to some variable that affects i only indirectly.

How are these competing considerations to be weighed? I have no idea. But the framework we have developed is interesting because it allows us to work on this question. In more basic terms, we can now "debug" partial solutions to CSPs directly, moving laterally through the search space in an attempt to remain as close to a solution as possible. This sort of lateral movement seems central to human solution of difficult search problems, and it is encouraging to begin to understand it in a formal way.

Dependency pruning

It is often the case that when one value for a variable is eliminated while solving a CSP, others are eliminated as well. As an example, in solving a scheduling problem a particular choice of time (say $t = 16$) may be eliminated for a task A because there then isn't enough time between A and a subsequent task B; in this case, all later times can obviously be eliminated for A as well.

Formalizing this can be subtle; after all, a later time for A isn't *uniformly* worse than an earlier time because there may be other tasks that need to precede A and making A later makes that part of the schedule easier. It's the problem with B alone that forces A to be earlier; once again, the analysis depends on the ability to maintain dependency information as the search proceeds.

We can formalize this as follows. Given a CSP (I, V, κ), suppose that the value v has been assigned to some $i \in I$. Now we can construct a new CSP (I', V', κ') involving the remaining variables $I' = I - \{i\}$, where the new set V' need not mention the possible values V_i for i, and where κ' is generated from κ by modifying the constraints to indicate that i has been assigned the value v. We also make the following definition:

Definition 2.7.1 *Given a CSP, suppose that i is a variable that has two possible* *values u and v. We will say that v is* stricter *than u if every constraint in the* CSP *induced by assigning u to i is also a constraint in the* CSP *induced by assigning i the value v.*

The point, of course, is that if v is stricter than u is, there is no point to trying a solution involving v once u has been eliminated. After all, finding such a solution would involve satisfying all of the constraints in the v restriction, these are a superset of those in the u restriction, and we were unable to satisfy the constraints in the u restriction originally.

The example with which we began this section now generalizes to the following:

Proposition 2.7.2 *Suppose that a* CSP *involves a set S of variables, and that we have a partial solution that assigns values to the variables in some subset $P \subseteq S$. Suppose further that if we extend this partial solution by assigning the value u to a variable $i \notin P$, there is no further extension to a solution of the entire* CSP*. Now consider the* CSP *involving the variables in $S - P$ that is induced by the choices of values for variables in P. If v is stricter than u as a choice of value for i in this problem, the original* CSP *has no solution that both assigns v to i and extends the given partial solution on P.* ■

This proposition isn't quite enough; in the earlier example, the choice of $t = 17$ for A will not be stricter than $t = 16$ if there is any task that needs to be scheduled before A is. We need to record the fact that B (which is no longer assigned a value) is the source of the difficulty. To do this, we need to augment the dependency information with which we are working.

More precisely, when we say that a set of variables $\{x_i\}$ eliminates a value v for a variable x, we mean that our search to date has allowed us to conclude that

$$(v_1 = x_1) \wedge \cdots \wedge (v_k = x_k) \supset v \neq x$$

where the v_i are the current choices for the x_i. We can obviously rewrite this as

$$(v_1 = x_1) \wedge \cdots \wedge (v_k = x_k) \wedge (v = x) \supset F \tag{2.5}$$

where F indicates that the CSP in question has no solution.

Let's be more specific still, indicating in (2.5) exactly *which* CSP has no solution:

$$(v_1 = x_1) \wedge \cdots \wedge (v_k = x_k) \wedge (v = x) \supset F(I) \tag{2.6}$$

where I is the set of variables in the complete CSP.

Now we can address the example with which we began this section; the CSP that is known to fail in an expression such as (2.6) is not the entire problem, but only a subset of it. In the example, we are considering, the subproblem involves only the two tasks A and B. In general, we can augment our nogoods to include information about the subproblems on which they fail, and then measure strictness with respect to these restricted subproblems only. In our example, this will indeed allow us to eliminate $t = 17$ from consideration as a possible time for A.

The additional information stored with the nogoods doubles their size (we have to store a second subset of the variables in the CSP), and the variable sets involved can be manipulated easily as the search proceeds. The cost involved in employing this technique is therefore that of the strictness computation. This may be substantial given the data structures currently used to represent CSPs (which typically support the need to check if a constraint has been violated but little more), but it seems likely that compile-time modifications to these data structures can be used to make the strictness question easier to answer. In scheduling problems, preliminary experimental work shows that the idea is an important one; here, too, there is much to be done.

The basic lesson of dynamic backtracking is that by retaining only those nogoods that are still relevant given the partial solution with which we are working, the storage difficulties encountered by full dependency-directed methods can be alleviated. This is what makes all of the ideas we have proposed possible – erasing values, selecting alternate backtrack points, and dependency pruning. There are surely many other effective uses for a practical dependency maintenance system as well.

2.8 Proofs

Lemma 2.2.4 *Let ε be a complete elimination mechanism for a CSP, let P be a partial solution to this CSP and let $i \notin \overline{P}$. Now if P can be successfully extended to a complete solution after assigning i the value v, then $v \notin \widehat{\varepsilon}(P, i)$.*

Proof. Suppose otherwise, so that $(v, E) \in \varepsilon(P, i)$. It follows directly from the completeness of ε that

$$E \cap (\overline{P} - \overline{P}) \neq \emptyset$$

a contradiction. ■

Lemma 2.2.6 *At any point in the execution of Algorithm 2.2.5, if the last element of the partial solution P assigns a value to the variable i, then the unexplored siblings of the current node are those that assign to i the values in $V_i - E_i$.*

Proof. We first note that when we decide to assign a value to a new variable i in step 2 of the algorithm, we take $E_i = \widehat{\varepsilon}(P, i)$ so that $V_i - E_i$ is the set of allowed values for this variable. The lemma therefore holds in this case. The fact that it continues to hold through each repetition of the loop in steps 3 and 4 is now a simple induction; at each point, we add to E_i the node that has just failed as a possible value to be assigned to i. ■

Proposition 2.2.7 *Algorithm 2.2.5 is equivalent to depth-first search and therefore complete.*

Proof. This is an easy consequence of the lemma. Partial solutions correspond to nodes in the search space. ■

Lemma 2.3.2 *Let P be a partial solution obtained during the execution of Algorithm 2.3.1, and let $i \in \overline{P}$ be a variable assigned a value by P. Now if $P' \subseteq P$ can be successfully extended to a complete solution after assigning i the value v but $(v, E) \in E_i$, we must have*

$$E \cap (\overline{P} - \overline{P'}) \neq \emptyset$$

Proof. As in the proof of Lemma 2.2.6, we show that no step of Algorithm 2.3.1 can cause Lemma 2.3.2 to become false.

That the lemma holds after step 2, where the search is extended to consider a new variable, is an immediate consequence of the assumption that the elimination mechanism is complete.

In step 4, when we add $(v_j, E - \{j\})$ to the set of eliminating explanations for j, we are simply recording the fact that the search for a solution with j set to v_j failed because we were unable to extend the solution to i. It is a consequence of the inductive hypothesis that as long as no variable in $E - \{j\}$ changes, this conclusion will remain valid. ■

Proposition 2.3.4 *Backjumping is complete and always expands fewer nodes than does depth-first search.*

Proof. That fewer nodes are examined is clear; for completeness, it follows from Lemma 2.3.2 that the backtrack to some element of E in step 5 will always be necessary if a solution is to be found. ■

Proposition 2.3.5 *The amount of space needed by backjumping is $o(i^2 v)$, where $i = |I|$ is the number of variables in the problem and v is the number of values for that variable with the largest value set V_i.*

Proof. The amount of space needed is dominated by the storage requirements of the elimination sets E_j; there are i of these. Each one might refer to each of the possible values for a particular variable j; the space needed to store the reason that the value j is eliminated is at most $|I|$, since the reason is simply a list of variables that have been assigned values. There will never be two eliminating explanations for the same variable, since ε is concise and we never rebind a variable to a value that has been eliminated. ■

Theorem 2.4.2 *Dynamic backtracking always terminates and is complete. It continues to satisfy Proposition 2.3.5 and can be expected to expand fewer nodes than backjumping provided that the goal nodes are distributed randomly in the search space.*

Proof. There are four things we need to show: That dynamic backtracking needs $o(i^2 v)$ space, that it is complete, that it can be expected to expand fewer nodes than backjumping, and that it terminates. We prove things in this order.

Space This is clear; the amount of space needed continues to be bounded by the structure of the eliminating explanations.

Completeness This is also clear, since by Lemma 2.3.2, all of the eliminating explanations retained in the algorithm are obviously still valid. The new explanations added in (2.2) are also obviously correct, since they indicate that j cannot take the value v_j as in backjumping and that j also cannot take any values that are eliminated by the variables being backjumped over.

Efficiency To see that we *expect* to expand fewer nodes, suppose that the subproblem involving only the variables being jumped over has s solutions in total, one of which is given by the existing variable assignments. Assuming that the solutions are distributed randomly in the search space, there is at least a $1/s$ chance that this particular solution leads to a solution of the entire CSP; if so, the reordered search – which considers this solution earlier than the other – will save the expense of either assigning new values to these variables or repeating the search that led to the existing choices. The reordered search will also benefit from the information in the nogoods that have been retained for the variables being jumped over.

Termination This is the most difficult part of the proof.

As we work through the algorithm, we will be generating (and then discarding) a variety of eliminating explanations. Suppose that e is such an explanation, saying that j cannot take the value v_j because of the values currently taken by the variables in some set e_V. We will denote the variables in e_V by $x_1, \ldots, x_k$ and their current values by $v_1, \ldots, v_k$. In declarative terms, the eliminating explanation is telling us that

$$(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k) \supset j \neq v_j \tag{2.7}$$

Dependency-directed backtracking would have us accumulate all of these nogoods; dynamic backtracking allows us to drop any particular instance of (2.7) for which the antecedent is no longer valid.

The reason that *dependency-directed* backtracking is guaranteed to terminate is that the set of accumulated nogoods eliminates a monotonically increasing amount of the search space. Each nogood eliminates a new section of the search space because the nature of the search process is such that any node examined is consistent with the nogoods that have been accumulated thus far; the process is monotonic because all nogoods are retained throughout the search. These arguments cannot be applied to dynamic backtracking, since nogoods are forgotten as the search proceeds. But we can make an analogous argument.

To do this, suppose that when we discover a nogood like (2.7), we record with it all of the variables that precede the variable j in the partial order, together with the values currently assigned to these variables. Thus an eliminating explanation becomes essentially a nogood n of the form (2.7) together with a set S of variable/value pairs.

We now define a mapping $\lambda(n, S)$ that changes the antecedent of (2.7) to include assumptions about *all* the variables bound in S, so that if $S = \{s_i, v_i\}$,

$$\lambda(n, S) = [(s_1 = v_1) \wedge \cdots \wedge (s_l = v_l) \supset j \neq v_j] \tag{2.8}$$

At any point in the execution of the algorithm, we denote by N the conjunction of the modified nogoods of the form (2.8).

We now make the following claims:

1. For any eliminating explanation (n, S), $n \models \lambda(n, S)$ so that $\lambda(n, S)$ is valid for the problem at hand.

2. For any new eliminating explanation (n, S), $\lambda(n, S)$ is not a consequence of N.

3. The deductive consequences of N grow monotonically as the dynamic backtracking algorithm proceeds.

The theorem will follow from these three observations, since we will know that N is a valid set of conclusions for our search problem and that we are once again making monotonic progress toward eliminating the entire search space and concluding that the problem is unsolvable.

That $\lambda(n, S)$ is a consequence of (n, S) is clear, since the modification used to obtain (2.8) from (2.7) involves strengthening that antecedent of (2.7). It is also clear that $\lambda(n, S)$ is not a consequence of the nogoods already obtained, since we have added to the antecedent only conditions that hold for the node of the search space currently under examination. If $\lambda(n, S)$ were a consequence of the nogoods we had obtained thus far, this node would not be being considered.

The last observation depends on the following lemma:

Lemma 2.8.1 *Suppose that x is a variable assigned a value by our partial solution and that x appears in the antecedent of the nogood n in the pair (n, S). Then if S' is the set of variables assigned values no later than x, $S' \subseteq S$.*

Proof. Consider a $y \in S'$, and suppose that it were not in S. We cannot have $y = x$, since y would then be mentioned in the nogood n and therefore in S. So we can suppose that y is actually assigned a value *earlier* than x is. Now when (n, S) was added to the set of eliminating explanations, it must have been the case that x was assigned a value (since it appears in the antecedent of n) but that y was not. But we also know that there was a later time when y was assigned a value but x was not, since y precedes x in the current partial solution. This means that x must have changed value at some point after (n, S) was added to the set of eliminating explanations – but (n, S) would have been deleted when this happened. This contradiction completes the proof. ■

Returning to the proof the Theorem 2.4.2, suppose that we eventually drop (n, S) from our collection of nogoods and that when we do so, the new nogood being added is (n', S'). It follows from the lemma that $S' \subseteq S$. Since $x_i = v_i$ is a clause in the antecedent of $\lambda(n, S)$, it follows that $\lambda(n', S')$ will imply the negation of the antecedent of $\lambda(n, S)$ and will therefore imply $\lambda(n, S)$ itself. Although we drop $\lambda(n, S)$ when we drop the nogood (n, S), $\lambda(n, S)$ continues to be entailed by the modified set N, the consequences of which are seen to be growing monotonically. ■

Chapter 3

Partial Order Dynamic Backtracking

Dynamic backtracking is the first search algorithm to allow branch variables to be reordered as the search progresses. However, it is still limited in an important way: backtracking remains chronological. By this we mean that when an inconsistency is detected, the chronologically most recent variable is always the one whose value is changed. This means that if an "early mistake" is made (*i.e.,* if a variable early in the search tree is given an incorrect value) the entire subtree under this "mistake" will have to be searched before the variable is revalued. Partial order dynamic backtracking changes this. Variables anywhere in the search tree can be revalued at any point. There is one catch though: each backtrack point adds to an evolving partial order on the set of variables, and this partial order eventually does constrain the order in which variables can be reordered (the partial order can be over-ridden only at the cost of an increase in the storage space needed).

This material appeared in "GSAT and dynamic backtracking" by Ginsberg and McAllester, in *KR-94*.

3.1 Introduction

The past few years have seen rapid progress in the development of algorithms for solving constraint-satisfaction problems, or CSPs. CSPs arise naturally in subfields of AI from planning to vision, and examples include propositional theorem proving, map coloring and scheduling problems. The problems are difficult because they involve search; there is never a guarantee that (for example) a successful coloring of a portion of a large map can be extended to a coloring of the map in its entirety.

The algorithms developed recently have been of two types. *Systematic* algorithms determine whether a solution exists by searching the entire space. *Local* algorithms use hill-climbing techniques to find a solution quickly but are *nonsystematic* in that they search the entire space in only a probabilistic sense.

The empirical effectiveness of these nonsystematic algorithms appears to be a result of their ability to follow local gradients in the search space. Traditional systematic procedures explore the space in a fixed order that is independent of local gradients; the fixed order makes following local gradients impossible but is needed to ensure that no node is examined twice and that the search remains systematic.

Dynamic backtracking [25] attempts to overcome this problem by retaining specific information about those portions of the search space that have been eliminated and then following local gradients in the remainder. Unlike previous algorithms that recorded such elimination information, such as dependency-directed backtracking [70], dynamic backtracking is selective about the information it caches so that only a polynomial amount of memory is required. These earlier techniques cached a new result with every backtrack, using an amount of memory that was linear in the run time and thus exponential in the size of the problem being solved.

Unfortunately, neither dynamic nor dependency-directed backtracking (or any other known similar method) is truly effective at local maneuvering within the search space, since the basic underlying methodology remains simple chronological backtracking. New techniques are included to make the search more efficient, but an exponential number of nodes in the search space must still be examined before early choices can be retracted. No existing search technique is able to both move freely within the search space and keep track of what has been searched and what hasn't.

The second class of algorithms developed recently presume that freedom of movement is of greater importance than systematicity. Algorithms in this class achieve their freedom of movement by abandoning the conventional description of the search space as a tree of partial solutions, instead thinking of it as a space of total assignments of values to variables. Motion is permitted between any two assignments that differ on a single value, and a hill-climbing procedure is employed to try to minimize the number of constraints violated by the overall assignment. The best-known algorithms in this class are min-conflicts [50] and GSAT [63].

Min-conflicts has been applied to the scheduling domain specifically and used to schedule tasks on the Hubble space telescope. GSAT is restricted to Boolean satisfiability problems (where every variable is assigned simply true or false), and has led to remarkable progress in the solution of randomly generated problems of this type; its performance is reported [59, 63, 60] as surpassing that of other techniques such as simulated annealing [41] and systematic techniques based on the Davis-Putnam procedure [15].

GSAT is not a panacea, however; there are many problems on which it performs fairly poorly. If a problem has no solution, for example, GSAT will never be able to report this with confidence. Even if a solution does exist, there appear to be at least two possible difficulties that GSAT may encounter.

First, the GSAT search space may contain so many local minima that it is not clear how GSAT can move so as to reduce the number of constraints violated by a given assignment. As an example, consider the CSP of generating crossword puzzles by filling words from a fixed dictionary into an empty frame [28]. The constraints indicate that there must be no conflict in each of the squares; thus two words that begin on the same square must also begin with the same letter. In this domain, getting "close" is not necessarily any indication that the problem is nearly solved, since correcting a conflict at a single square may involve modifying much of the current solution. Konolige has recently reported that GSAT specifically has difficulty solving problems of this sort [42].

Second, GSAT does no forward propagation. In the crossword domain once again, selecting one word may well force the selection of a variety of subsequent words. In a Boolean satisfiability problem, assigning one variable the value true may cause an immediate cascade of values to be assigned to other variables via a technique known as *unit resolution.* It seems plausible that forward propagation will be more common on realistic problems than on randomly generated ones; the most difficult random problems appear to be tangles of closely related individual variables while naturally occurring problems tend to be tangles of sequences of related variables. Furthermore, it appears that GSAT's performance degrades (relative to systematic approaches) as these sequences of variables arise [12].

Our aim in this paper is to describe a new search procedure that appears to combine the benefits of both of the earlier approaches; in some very loose sense, it can be thought of as a systematic version of GSAT.

The next three sections summarize the original dynamic backtracking algorithm [25], presenting it from the perspective of local search. The termination proof is omitted here but can be found in earlier papers [25, 49]. Section 3.5 present a modification of dynamic backtracking called *partial-order dynamic backtracking,* or PDB. This algorithm builds on work of McAllester's [49]. Partial-order dynamic backtracking provides greater flexibility in the allowed set of search directions while preserving systematicity and polynomial worst case space usage. Section 3.6 presents a new variant of dynamic backtracking that is still more flexible in the allowed set of search directions. While this final procedure is still systematic, it can use exponential space in the worst case. Section 3.7 presents some empirical results comparing PDB with other well known algorithms on a class of "local" randomly generated 3-SAT problems. Concluding remarks are contained in Section 3.8, and proofs appear in the last section.

3.2 Constraints and Nogoods

We begin with a slightly nonstandard definition of a CSP.

Definition 3.2.1 *By a* constraint satisfaction problem (I, V, κ) *we will mean a finite set I of variables; for each $x \in I$, there is a finite set V_x of possible values for the variable x. κ is a set of constraints each of the form $\neg[(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k)]$ where each x_j is a variable in I and each v_j is an element of V_{x_j}. A* solution *to the* CSP *is an* *assignment P of values to variables that satisfies every constraint. For each variable x we require that $P(x) \in V_x$ and for each constraint $\neg[(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k)]$ we require that $P(x_i) \neq v_i$ for some x_i.*

By the size *of a constraint-satisfaction problem (I, V, κ), we will mean the product of the domain sizes of the various variables, $\prod_x |V_x|$.*

The technical convenience of the above definition of a constraint will be clear shortly. For the moment, we merely note that the above description is clearly equivalent to the conventional one; rather than represent the constraints in terms of allowed value combinations for various variables, we write axioms that disallow specific value combinations one at a time. The size of a CSP is the number of possible assignments of values to variables.

Systematic algorithms attempting to find a solution to a CSP typically work with partial solutions that are then discovered to be inextensible or to violate the given constraints; when this happens, a backtrack occurs and the partial solution under consideration is modified. Such a procedure will, of course, need to record information that guarantees that the same partial solution not be considered again as the search proceeds. This information might be recorded in the structure of the search itself; depth-first search with chronological backtracking is an example. More sophisticated methods maintain a database of some form indicating explicitly which choices have been eliminated and which have not. In this paper, we will use a database consisting of a set of *nogoods* [17].

Definition 3.2.2 *A* nogood *is an expression of the form*

$$(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k) \rightarrow x \neq v \tag{3.1}$$

A nogood can be used to represent a constraint as an implication; (3.1) is logically equivalent to the constraint

$$\neg[(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k) \wedge (x = v)]$$

There are clearly many different ways of representing a given constraint as a nogood.

One special nogood is the *empty* nogood, which is tautologically false. We will denote the empty nogood by $\perp$; if $\perp$ can be derived from the given set of constraints, it follows that no solution exists for the problem being attempted.

The typical way in which new nogoods are obtained is by resolving together old ones. As an example, suppose we have derived the following:

$$\begin{aligned} (x = a) \wedge (y = b) &\rightarrow u \neq v_1 \\ (x = a) \wedge (z = c) &\rightarrow u \neq v_2 \\ (y = b) &\rightarrow u \neq v_3 \end{aligned}$$

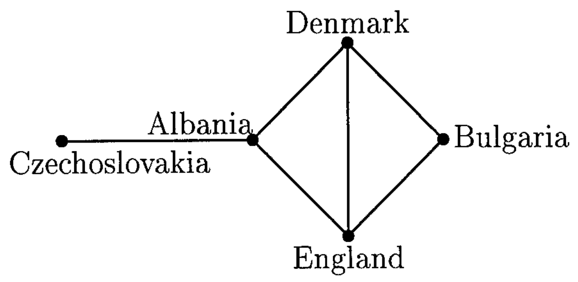

Figure 3.1: A small map-coloring problem

where v_1, v_2 and v_3 are the only values in the domain of u. It follows that we can combine these nogoods to conclude that there is no solution with

$$(x = a) \wedge (y = b) \wedge (z = c) \tag{3.2}$$

Moving z to the conclusion of (3.2) gives us

$$(x = a) \wedge (y = b) \rightarrow z \neq c$$

In general, suppose we have a collection of nogoods of the form

$$x_{i1} = v_{i1} \wedge \cdots \wedge x_{in_i} = v_{in_i} \rightarrow x \neq v_i$$

as i varies, where the same variable appears in the conclusions of all the nogoods. Suppose further that the antecedents all agree as to the value of the x_i's, so that any time x_i appears in the antecedent of one of the nogoods, it is in a term $x_i = v_i$ for a fixed v_i. If the nogoods collectively eliminate all of the possible values for x, we can conclude that $\bigwedge_j (x_j = v_j)$ is inconsistent; moving one specific x_k to the conclusion gives us

$$\bigwedge_{j \neq k} (x_j = v_j) \rightarrow x_k \neq v_k \tag{3.3}$$

As before, note the freedom in our choice of variable appearing in the conclusion of the nogood. Since the next step in our search algorithm will presumably satisfy (3.3) by changing the value for x_k, the selection of consequent variable corresponds to the choice of variable to "flip" in the terms used by GSAT or other hill-climbing algorithms.

As we have remarked, dynamic backtracking accumulates information in a set of nogoods. To see how this is done, consider the map coloring problem in Figure 3.1, repeated from [25]. The map consists of five countries: Albania, Bulgaria, Czechoslovakia, Denmark and England. We assume – wrongly – that the countries border each other as shown in the figure, where countries are denoted by nodes and border one another if and only if there is an arc connecting them.

In coloring the map, we can use the three colors red, green and blue. We will typically abbreviate the colors and country names to single letters in the obvious way. The following table gives a trace of how a conventional dependency-directed backtracking scheme might attack this problem; each row shows a state of the procedure in the middle of a backtrack step, after a new nogood has been identified but before colors are erased to reflect the new conclusion. The coloring that is about to be removed appears in boldface. The "drop" column will be discussed shortly.

A	B	C	D	E	add		drop
r	g	**r**			$A = r \rightarrow C \neq r$	1	
r	g	b	**r**		$A = r \rightarrow D \neq r$	2	
r	g	b	**g**		$B = g \rightarrow D \neq g$	3	
r	g	b	b	**r**	$A = r \rightarrow E \neq r$	4	
r	g	b	b	**g**	$B = g \rightarrow E \neq g$	5	
r	g	b	b	**b**	$D = b \rightarrow E \neq b$	6	
r	g	b	**b**		$(A = r) \wedge (B = g) \rightarrow D \neq b$	7	6
r	**g**	b			$A = r \rightarrow B \neq g$	8	$3, 5, 7$

We begin by coloring Albania red and Bulgaria green, and then try to color Czechoslovakia red as well. Since this violates the constraint that Albania and Czechoslovakia be different colors, nogood (1) in the above table is produced.

We change Czechoslovakia's color to blue and then turn to Denmark. Since Denmark cannot be colored red or green, nogoods (2) and (3) appear; the only remaining color for Denmark is blue.

Unfortunately, having colored Denmark blue, we cannot color England. The three nogoods generated are (4), (5) and (6), and we can resolve these together because the three conclusions eliminate all of the possible colors for England. The result is that there is no solution with $(A = r) \wedge (B = g) \wedge (D = b)$, which we rewrite as (7) above. This can in turn be resolved with (2) and (3) to get (8), correctly indicating that the color of red for Albania is inconsistent with the choice of green for Bulgaria. The analysis can continue at this point to gradually determine that Bulgaria has to be red, Denmark can be green or blue, and England must then be the color not chosen for Denmark.

As we mentioned in the introduction, the problem with this approach is that the set Γ of nogoods grows monotonically, with a new nogood being added at every step. The number of nogoods stored therefore grows linearly with the run time and thus (presumably) exponentially with the size of the problem. A related problem is that it may become increasingly difficult to extend the partial solution P without violating one of the nogoods in Γ.

Dynamic backtracking deals with this by discarding nogoods when they become "irrelevant" in the sense that their antecedents no longer match the partial solution in question. In the example above, nogoods can be eliminated as indicated in the final column of the trace. When we derive (7), we remove (6) because Denmark is no longer colored blue. When we derive (8), we remove all of the nogoods with $B = g$ in their antecedents. Thus the only information we retain is that Albania's red color precludes red for Czechoslovakia, Denmark and England (1, 2 and 4) and also green for Bulgaria (8).

3.3 Dynamic Backtracking

Dynamic backtracking uses the set of nogoods to both record information about the portion of the search space that has been eliminated and to record the current partial assignment being considered by the procedure. The current partial assignment is encoded in the antecedents of the current nogood set. More formally:

Definition 3.3.1 *An* acceptable next assignment *for a nogood set* Γ *is an assignment* P *satisfying every nogood in* Γ *and every antecedent of every such nogood. We will call a set of nogoods* Γ acceptable *if no two nogoods in* Γ *have the same conclusion and either* $\bot \in \Gamma$ *or there exists an acceptable next assignment for* Γ.

If Γ is acceptable, the antecedents of the nogoods in Γ induce a partial assignment of values to variables; any acceptable next assignment must be an extension of this partial assignment. In the above table, for example, nogoods (1) through (6) encode the partial assignment given by $A = r$, $B = g$, and $D = b$. Nogoods (1) though (7) fail to encode a partial assignment because the seventh nogood is inconsistent with the partial assignment encoded in nogoods (1) through (6). This is why the sixth nogood is removed when the seventh nogood is added.

Procedure 3.3.2 (Dynamic backtracking) To solve a CSP:

```
$P :=$ any complete assignment of values to variables
$\Gamma := \emptyset$
**until** either $P$ is a solution or $\bot \in \Gamma$:
    $\gamma :=$ any constraint violated by $P$
    $\Gamma := \texttt{simp}(\Gamma \cup \gamma)$
    $P :=$ any acceptable next assignment for $\Gamma$
```

To simplify the discussion we assume a fixed total order on the variables. Versions of dynamic backtracking with dynamic rearrangement of the variable order can be found elsewhere [25, 49]. Whenever a new nogood is added, the fixed variable ordering is used to select the variable that appears in the conclusion of the nogood – the latest variable always appears in the conclusion. The subroutine `simp` closes the set of nogoods under the resolution inference rule discussed in the previous section and removes all nogoods which have an antecedent $x = v$ such that $x \neq v$ appears in the conclusion of some other nogood. Without giving a detailed analysis, we note that simplification ensures that Γ remains acceptable. To prove termination we introduce the following notation:

Definition 3.3.3 *For any acceptable Γ and variable x, we define the* live domain of x *to be those values v such that $x \neq v$ does not appear in the conclusion of any nogood in Γ. We will denote the size of the live domain of x by $|x|_\Gamma$, and will denote by $m(\Gamma)$ the tuple $\langle |x_1|_\Gamma, \ldots, |x_n|_\Gamma \rangle$ where $x_1, \ldots, x_n$ are the variables in the* CSP *in their specified order.*

Given an acceptable Γ, we define the size of Γ *to be*

$$\texttt{size}(\Gamma) = \prod_x |V_x| - \sum_x \left[(|V_x| - |x|_\Gamma) \prod_{x_i > x} |V_{x_i}| \right]$$

Informally, the size of Γ is the size of the remaining search space given the live domains for the variables and assuming that all information about x_i will be lost when we change the value for any variable $x_j < x_i$.

The following result is obvious:

Lemma 3.3.4 *Suppose that Γ and Γ' are such that $m(\Gamma)$ is lexicographically less than $m(\Gamma')$. Then* $\texttt{size}(\Gamma) < \texttt{size}(\Gamma')$. ■

The termination proof (which we do not repeat here) is based on the observation that every simplification lexicographically reduces $m(\Gamma)$. Assuming that $\Gamma = \emptyset$ initially, since

$$\texttt{size}(\emptyset) = \prod_x |V_x|$$

it follows that the running time of dynamic backtracking is bounded by the size of the problem being solved.

Proposition 3.3.5 *Any acceptable set of nogoods can be stored in $o(n^2 v)$ space where n is the number of variables and v is the maximum domain size of any single variable.*

It is worth considering the behavior of Procedure 3.3.2 when applied to a CSP that is the union of two disjoint CSPs that do not share variables or constraints. If each of the two subproblems is unsatisfiable and the variable ordering interleaves the variables of the two subproblems, a classical backtracking search will take time proportional to the product of the times required to search each assignment space separately.[1] In

[1]This observation remains true even if backjumping techniques are used.

contrast, Procedure 3.3.2 works on the two problems independently, and the time taken to solve the union of problems is therefore the sum of the times needed for the individual subproblems. It follows that Procedure 3.3.2 is fundamentally different from classical backtracking or backjumping procedures; Procedure 3.3.2 is in fact what has been called a *polynomial space aggressive backtracking procedure* [49].

3.4 Dynamic Backtracking as Local Search

Before proceeding, let us highlight the obvious similarities between Procedure 3.3.2 and Selman's description of GSAT [63]:

Procedure 3.4.1 (GSAT) To solve a CSP:

```
for $i$ := 1 to MAX-TRIES
    $P$ := a randomly generated truth assignment
    for $j$ := 1 to MAX-FLIPS
        if $P$ is a solution, then return it
        else flip any variable in $P$ that results in
                  the greatest decrease in the number
                  of unsatisfied clauses
        end if
    end for
end for
return failure
```

The inner loop of the above procedure makes a local move in the search space in a direction consistent with the goal of satisfying a maximum number of clauses; we will say that GSAT follows the local gradient of a "maxsat" objective function. But local search can get stuck in local minima; the outer loop provides a partial escape by giving the procedure several independent chances to find a solution.

Like GSAT, dynamic backtracking examines a sequence of total assignments. Initially, dynamic backtracking has considerable freedom in selecting the next assignment; in many cases, it can update the total assignment in a manner identical to GSAT. The nogood set ultimately both constrains the allowed directions of motion and forces the procedure to search systematically. Dynamic backtracking cannot get stuck in local minima.

Both systematicity and the ability to follow local gradients are desirable. The observations of the previous paragraphs, however, indicate that these two properties are in conflict – systematic enumeration of the search space appears incompatible with gradient descent. To better understand the interaction of systematicity and local gradients, we need to examine more closely the structure of the nogoods used in dynamic backtracking.

We have already discussed the fact that a single constraint can be represented as a nogood in a variety of ways. For example, the constraint $\neg(A = r \wedge B = g)$ can be represented either as $A = r \rightarrow B \neq g$ or as $B = g \rightarrow A \neq r$. Although these nogoods capture the same information, they behave differently in the dynamic backtracking procedure because they encode different partial truth assignments and represent different choices of variable ordering. In particular, the set of acceptable next assignments for $A = r \rightarrow B \neq g$ is quite different from the set of acceptable next assignments for $B = g \rightarrow A \neq r$. In the former case an acceptable assignment must satisfy $A = r$; in the latter case, $B = g$ must hold. Intuitively, the former nogood corresponds to changing the value of B while the latter nogood corresponds to changing that of A. The manner in which we represent the constraint $\neg(A = r \wedge B = g)$ influences the direction in which the search is allowed to proceed. In Procedure 3.3.2, the choice of representation is forced by the need to respect the fixed variable ordering and to change the latest variable in the constraint.[2] Similar restrictions exist in the original presentation of dynamic backtracking itself [25].

3.5 Partial-order Dynamic Backtracking

Partial-order dynamic backtracking [49] replaces the fixed variable order with a *partial* order that is dynamically modified during the search. When a new nogood is added, this partial ordering need not fix a unique representation – there can be considerable choice in the selection of the variable to appear in the conclusion of the nogood. This leads to freedom in the selection of the variable whose value is to be changed, thereby allowing greater flexibility in the directions that the procedure can take while traversing the search space. The locally optimal gradient followed by GSAT can be adhered to more often. The partial order on variables is represented by a set of ordering constraints called *safety conditions.*

Definition 3.5.1 *A* safety condition *is an assertion of the form $x < y$ where x and y are variables. Given a set S of safety conditions, we will denote by $\leq_S$ the transitive closure of $<$, saying that S is* acyclic *if $\leq_S$ is antisymmetric. We will write $x <_S y$ to mean that $x \leq_S y$ and $y \not\leq_S x$.*

In other words, $x \leq y$ if there is some (possibly empty) sequence of safety conditions

$$x < z_1 < \ldots < z_n < y$$

The requirement of antisymmetry means simply that there are no two distinct x and y for which $x \leq y$ and $y \leq x$; in other words, $\leq_S$ has no "loops" and is a partial order

[2]Note, however, that there is still considerable freedom in the choice of the constraint itself. A total assignment usually violates many different constraints.

on the variables. In this section, we restrict our attention to acyclic sets of safety conditions.

Definition 3.5.2 *For a nogood γ, we will denote by S_γ the set of all safety conditions $x < y$ such that x is in the antecedent of γ and y is the variable in its conclusion.*

Informally, we require variables in the antecedent of nogoods to precede the variables in their conclusions, since the antecedent variables have been used to constrain the live domains of the conclusions.

The state of the partial order dynamic backtracking procedure is represented by a pair $\langle \Gamma, S \rangle$ consisting of a set of nogoods and a set of safety conditions. In many cases, we will be interested in only the ordering information about variables that can precede a fixed variable x. To discard the rest of the ordering information, we discard all of the safety conditions involving any variable y that follows x, and then record only that y does indeed follow x. Somewhat more formally:

Definition 3.5.3 *For any set S of safety conditions and variable x, we define the* weakening of S at x, *to be denoted $W(S, x)$, to be the set of safety conditions given by removing from S all safety conditions of the form $z < y$ where $x <_S y$ and then adding the safety condition $x < y$ for all such y.*

The set $W(S, x)$ is a weakening of S in the sense that every total ordering consistent with S is also consistent with $W(S, x)$. However $W(S, x)$ usually admits more total orderings than S does; for example, if S specifies a total order then $W(S, x)$ allows any order which agrees with S up to and including the variable x. In general, we have the following:

Lemma 3.5.4 *For any set S of safety conditions, variable x, and total order $<$ consistent with the safety conditions in $W(S, x)$, there exists a total order consistent with S that agrees with $<$ through x.*

We now state the PDB procedure.

Procedure 3.5.5 To solve a CSP:

```
$P :=$ any complete assignment of values to variables
$\Gamma := \emptyset$
$S := \emptyset$
**until** either $P$ is a solution or $\bot \in \Gamma$:
    $\gamma :=$ a constraint violated by $P$
    $\langle \Gamma, S \rangle := \texttt{simp}(\Gamma, S, \gamma)$
    $P :=$ any acceptable next assignment for $\Gamma$
```

Procedure 3.5.6 To compute `simp`(Γ, S, γ):

```
select the conclusion $x$ of $\gamma$ so that $S \cup S_\gamma$ is acyclic
$\Gamma := \Gamma \cup \{\gamma\}$
$S := W(S \cup S_\gamma, x)$
remove from $\Gamma$ each nogood with $x$ in its antecedent
if the conclusions of nogoods in $\Gamma$ rule out all
        possible values for $x$ then
    $\rho :=$ the result of resolving all nogoods in $\Gamma$ with $x$
          in their conclusion
    $\langle \Gamma, S \rangle :=$ simp$(\Gamma, S, \rho)$
end if
return $\langle \Gamma, S \rangle$
```

The above simplification procedure maintains the invariant that Γ be acceptable and S be acyclic; in addition, the time needed for a single call to `simp` appears to grow significantly sublinearly with the size of the problem in question (see Section 3.7).

Theorem 3.5.7 *Procedure 3.5.5 terminates. The number of calls to* `simp` *is bounded by the size of the problem being solved.*

As an example, suppose that we return to our map-coloring problem. We begin by coloring all of the countries red except Bulgaria, which is green. The following table shows the total assignment that existed at the moment each new nogood was generated.

A	B	C	D	E	add		drop
r	g	r	r	r	$C = r \rightarrow A \neq r$	1	
b	g	r	r	r	$D = r \rightarrow E \neq r$	2	
b	g	r	r	g	$B = g \rightarrow E \neq g$	3	
b	g	r	r	b	$A = b \rightarrow E \neq b$	4	
					$(A = b) \wedge (B = g) \rightarrow D \neq r$	5	2
					$D < E$	6	
b	g	r	g	r	$B = g \rightarrow D \neq g$	7	
b	g	r	b	r	$A = b \rightarrow D \neq b$	8	
					$A = b \rightarrow B \neq g$	9	$3, 5, 7$
					$B < E$	10	6
					$B < D$	11	

The initial coloring violates a variety of constraints; suppose that we choose to work on one with Albania in its conclusion because Albania is involved in three violated constraints. We choose $C = r \to A \neq r$ specifically, and add it as (1) above.

We now modify Albania to be blue. The only constraint violated is that Denmark and England be different colors, so we add (2) to Γ. This suggests that we change the color for England; we try green, but this conflicts with Bulgaria. If we write the new nogood as $E = g \to B \neq g$, we will change Bulgaria to blue and be done. In the table above, however, we have made the less optimal choice (3), changing the coloring for England again.

We are now forced to color England blue. This conflicts with Albania, and we continue to leave England in the conclusion of the nogood as we add (4). This nogood resolves with (2) and (3) to produce (5), where we have once again made the worst choice and put D in the conclusion. We add this nogood to Γ and remove nogood (2), which is the only nogood with D in its antecedent. In (6) we add a safety condition indicating that D must continue to precede E. (This safety condition has been present since nogood (2) was discovered, but we have not indicated it explicitly until the original nogood was dropped from the database.)

We next change Denmark to green; England is forced to be red once again. But now Bulgaria and Denmark are both green; we have to write this new nogood (7) with Denmark in the conclusion because of the ordering implied by nogood (5) above. Changing Denmark to blue conflicts with Albania (8), which we have to write as $A = b \to D \neq b$. This new nogood resolves with (5) and (7) to produce (9).

We drop (3), (5) and (7) because they involve $B = g$, and introduce the two safety conditions (10) and (11). Since E follows B, we drop the safety condition $E < D$. At this point, we are finally forced to change the color for Bulgaria and the search continues.

It is important to note that the added flexibility of PDB over dynamic backtracking arises from the flexibility in the first step of the simplification procedure where the conclusion of the new nogood is selected. This selection corresponds to a selection of a variable whose value is to be changed.

As with the procedure in the previous section, when given a CSP that is a union of disjoint CSPs the above procedure will treat the two subproblems independently. The total running time remains the sum of the times required for the subproblems.

3.6 Arbitrary Movement

Partial-order dynamic backtracking still does not provide total freedom in the choice of direction through the search space. When a new nogood is discovered, the existing partial order constrains how we are to interpret that nogood – roughly speaking, we are forced to change the value of late variables before changing the values of their predecessors. The use of a partial order makes this constraint looser than previously, but it is still present. In this section, we allow cycles in the nogoods and safety conditions, thereby permitting arbitrary choice in the selection of the variable appearing in the conclusion of a new nogood.

The basic idea is the following: Suppose that we have introduced a loop into the variable ordering, perhaps by including the pair of nogoods $x \rightarrow \neg y$ and $y \rightarrow x$. Rather than rewrite one of these nogoods so that the same variable appears in the conclusion of both, we will view the (x, y) combination as a single variable that takes a value in the product set $V_x \times V_y$.

If x and y are variables that have been "combined" in this way, we can rewrite a nogood with (for example) x in its antecedent and y in its conclusion so that both x and y are in the conclusion. As an example, we can rewrite

$$x = v_x \wedge z = v_z \rightarrow y \neq v_y \tag{3.4}$$

as

$$z = v_z \rightarrow (x, y) \neq (v_x, v_y) \tag{3.5}$$

which is logically equivalent. We can view this as eliminating a particular value for the pair of variables (x, y).

Definition 3.6.1 *Let S be a set of safety conditions (possibly not acyclic). We will write $x \equiv_S y$ if $x \leq_S y$ and $y \leq_S x$. The equivalence class of x under $\equiv$ will be denoted $\langle x \rangle_S$. If γ is a nogood whose conclusion involves the variable x, we will denote by γ_S the result of moving to the conclusion of γ all terms involving members of $\langle x \rangle_S$. If Γ is a set of nogoods, we will denote by Γ_S is the set of nogoods of the form γ_S for $\gamma \in \Gamma$.*

It is not difficult to show that for any set S of safety conditions, the relation $\equiv_S$ is an equivalence relation. As an example of rewriting a nogood in the presence of ordering cycles, suppose that γ is the nogood (3.4) and let S be such that $\langle y \rangle_S = \{x, y\}$; now γ_S is given by (3.5).

Placing more than one literal in the conclusions of nogoods forces us to reconsider the notion of an acceptable next assignment:

Definition 3.6.2 *A* cyclically acceptable next assignment *for a nogood set Γ under a set S of safety conditions is a total assignment P of values to variables satisfying every nogood in Γ_S and every antecedent of every such nogood.*

We now define a third dynamic backtracking procedure. Note that $W(S, x)$ remains well defined even if S is not acyclic, since $W(S, x)$ drops ordering constraints only on variables y such that $x <_S y$.

Procedure 3.6.3 To solve a CSP:

```
$P :=$ any complete assignment of values to variables
$\Gamma := \emptyset$
$S := \emptyset$
until either $P$ is a solution or $\bot \in \Gamma$:
        $\gamma :=$ a constraint violated by $P$
        $\langle \Gamma, S \rangle :=$ simp$(\Gamma, S, \gamma)$
        $P :=$ any cyclically acceptable next assignment
                for $\Gamma$ under $S$
```

Procedure 3.6.4 To compute simp(Γ, S, γ):

```
select a conclusion $x$ for $\gamma$ (now unconstrained)
$\Gamma := \Gamma \cup \{\gamma\}$
$S := W(S \cup S_\gamma, x)$
remove from $\Gamma$ each nogood $\alpha$ with an element of $\langle x \rangle_S$
            in the antecedent of $\alpha_S$
if the conclusions of nogoods in $\Gamma_S$ rule out all
          possible values for the variables in $\langle x \rangle_S$ then
   $\rho :=$ the result of resolving all nogoods in $\Gamma_S$ whose
           conclusions involve variables in $\langle x \rangle_S$
   $\langle \Gamma, S \rangle :=$ simp$(\Gamma, S, \rho)$
   end if
return $\langle \Gamma, S \rangle$
```

If the conclusion is selected so that S remains acyclic, the above procedure is identical to the one in the previous section.

Proposition 3.6.5 *Suppose that we are working on a problem with n variables, that the size of the largest domain of any variable is v, and that we have constructed Γ and S using repeated applications of* simp. *If the largest equivalence class $\langle x \rangle_S$ contains d elements, the space required to store Γ is $o(n^2 v^d)$.*

If we have an equivalence class of d variables each of which has v possible values then the number of possible values of the "combined variable" is v^d. The above procedure can now generate a distinct nogood to eliminate each of the v^d possible values, and the space requirements of the procedure can therefore grow exponentially in the size of the equivalence classes. The time required to find a cyclically allowed next assignment can also grow exponentially in the size of the equivalence classes. We can address these difficulties by selecting in advance a bound for the largest allowed size of any equivalence class. In any event, termination is still guaranteed:

Theorem 3.6.6 *Procedure 3.6.3 terminates. The number of calls to* `simp` *is bounded by the size of the problem being solved.*

Selecting a variable to place in the conclusion of a new nogood corresponds to choosing the variable whose value is to be changed on the next iteration and is analogous to selecting the variable to flip in GSAT. Since the choice of conclusion is unconstrained in the above procedure, the procedure has tremendous flexibility in the way it traverses the search space. Like the procedures in the previous sections, Procedure 3.6.3 continues to solve combinations of independent subproblems in time bounded by the sum of the times needed to solve the subproblems individually.

Here are these ideas in use on a Boolean CSP with the constraints $a \to b$, $b \to c$ and $c \to \neg b$. As before, we present a trace and then explain it:

a	b	c	add to Γ		remove from Γ
t	f	f	$a \to b$	1	
t	t	f	$b \to c$	2	
t	t	t	$c \to \neg b$	3	
			$\neg a$	4	1
			$a < b$	5	

The first three nogoods are simply the three constraints appearing in the problem. Although the orderings of the second and third nogoods conflict, we choose to write them in the given form in any case.

Since this puts b and c into an equivalence class, we do not drop nogood (2) at this point. Instead, we interpret nogood (1) as requiring that the value taken by (b, c) be either (t, t) or (t, f); (2) disallows (t, f) and (3) disallows (t, t). It follows that the three nogoods can be resolved together to obtain the new nogood given simply by $\neg a$. We add this as (4) above, dropping nogood (1) because its antecedent is falsified.

3.7 Experimental Results

In this section, we present preliminary results regarding the implemented effectiveness of the procedure we have described. The implementation is based on the somewhat restricted Procedure 3.5.5 as opposed to the more general Procedure 3.6.3. We compared a search engine based on this procedure with two others, TABLEAU [11] and WSAT, or "walk-sat" [60]. TABLEAU is an efficient implementation of the Davis-Putnam algorithm and is systematic; WSAT is a modification to GSAT and is not. We used WSAT instead of GSAT because WSAT is more effective on a fairly wide range of problem distributions [60].

The experimental data was not collected using the random 3-SAT problems that have been the target of much recent investigation, since there is growing evidence that these problems are not representative of the difficulties encountered in practice [12]. Instead, we generated our problems so that the clauses they contain involve groups of locally connected variables as opposed to variables selected at random.

Somewhat more specifically, we filled an $n \times n$ square grid with variables, and then required that the three variables appearing in any single clause be neighbors in this grid. LISP code generating these examples appears in the appendix. We believe that the qualitative properties of the results reported here hold for a wide class of distributions where variables are given spatial locations and clauses are required to be local.

The experiments were performed at the crossover point where approximately half of the instances generated could be expected to be satisfiable, since this appears to be where the most difficult problems lie [11]. Note that not all instances at the crossover point are hard; as an example, the local variable interactions in these problems can lead to short resolution proofs that no solution exists in unsatisfiable cases. This is in sharp contrast with random 3-SAT problems (where no short proofs appear to exist in general, and it can even be shown that proof lengths are growing exponentially on average [7]). Realistic problems may often have short proof paths: A particular scheduling problem may be unsatisfiable simply because there is no way to schedule a specific resource as opposed to because of global issues involving the problem in its entirety. Satisfiability problems arising in VLSI circuit design can also be expected to have locality properties similar to those we have described.

The problems involved 25, 100, 225, 400 and 625 variables. For each size, we generated 100 satisfiable and 100 unsatisfiable instances and then executed the three procedures to measure their performance. (WSAT was not tested on the unsatisfiable instances.) For WSAT, we measured the number of times specific variable values were flipped. For PDB, we measured the number of top-level calls to Procedure 3.5.6. For TABLEAU, we measured the number of choice nodes expanded. WSAT and PDB were limited to 100,000 flips; TABLEAU was limited to a running time of 150 seconds.

The results for the satisfiable problems were as follows. For TABLEAU, we give the node count for successful runs only; we also indicate parenthetically what fraction of the problems were solved given the computational resource limitations. (WSAT and PDB successfully solved all instances.)

Variables	**PDB**	**WSAT**	**TABLEAU**
25	35	89	9 (1.0)
100	210	877	255 (1.0)
225	434	1626	504 (.98)
400	731	2737	856 (.70)
625	816	3121	502 (.68)

For the unsatisfiable instances, the results were:

Variables	PDB	TABLEAU
25	122	8 (1.0)
100	509	1779 (1.0)
225	988	5682 (.38)
400	1090	558 (.11)
625	1204	114 (.06)

The times required for PDB and WSAT appear to be growing comparably, although only PDB is able to solve the unsatisfiable instances. The eventual *decrease* in the average time needed by TABLEAU is because it is only managing to solve the easiest instances in each class. This causes TABLEAU to become almost completely ineffective in the unsatisfiable case and only partially effective in the satisfiable case. Even where it does succeed on large problems, TABLEAU's run time is greater than that of the other two methods.

Finally, we collected data on the time needed for each top-level call to `simp` in partial-order dynamic backtracking. As a function of the number of variables in the problem, this was:

Number of variables	PDB (msec)	WSAT (msec)
25	3.9	0.5
100	5.3	0.3
225	6.7	0.6
400	7.0	0.7
625	8.4	1.4

All times were measured on a Sparc 10/40 running unoptimized Allegro Common Lisp. An efficient C implementation could expect to improve either method by approximately an order of magnitude. As mentioned in Section 3.5, the time per flip is growing sublinearly with the number of variables in question.

3.8 Discussion

Our aim in this paper has been to make a primarily theoretical contribution, describing a new class of constraint-satisfaction algorithms that appear to combine many of the advantages of previous systematic and nonsystematic approaches. Since our focus has been on a description of the algorithms, there is obviously much that remains to be done.

First, of course, the procedures must be tested on a variety of problems, both synthetic and naturally occurring; the results reported in Section 3.7 only scratch the surface. It is especially important that realistic problems be included in any experimental evaluation of these ideas, since these problems are likely to have performance profiles substantially different from those of randomly generated problems [12]. The experiments of the previous section need to be extended to include unit resolution, and we need to determine the frequency with which exponential space is needed in practice by the full procedure 3.6.3.

Finally, we have left completely untouched the question of how the flexibility of Procedure 3.6.3 is to be exploited. Given a group of violated constraints, which should we pick to add to Γ? Which variable should be in the conclusion of the constraint? These choices correspond to choice of backtrack strategy in a more conventional setting, and it will be important to understand them in this setting as well.

3.9 Proofs

Proposition 3.3.5 *Any acceptable set of nogoods can be stored in $o(n^2 v)$ space where n is the number of variables and v is the maximum domain size of any single variable.*
Proof. This can be done by first storing the partial assignment encoded in Γ using $o(n)$ space. The antecedent of each nogood can now be represented as a bit vector specifying the set of variables appearing in the antecedent, allowing the nogood itself to be stored in $o(n)$ space. Since no two nogoods share the same conclusion there are at most nv nogoods. ■

Lemma 3.5.4 *For any set S of safety conditions, variable x and total order $<$ consistent with the safety conditions in $W(S,x)$, there is a total order consistent with S that agrees with $<$ through x.*
Proof. Suppose that the ordering $<$ is given by

$$x_1 < \cdots < x_k = x < y_1 < \cdots < y_m \tag{3.6}$$

Now let $<'$ be any ordering consistent with S, and suppose that the ordering given by $<'$ on the y_i in (3.6) is

$$z_1 <' \cdots <' z_m$$

We claim that the ordering given by

$$x_1, \ldots, x_k = x, z_1, \ldots, z_m \tag{3.7}$$

is consistent with all of S. We will show this by showing that (3.7) is consistent with any specific safety condition $u < v$ in S.

If both u and v are x_i's, then the safety condition $u < v$ will remain in $W(S,x)$ and is therefore satisfied by (3.7). If both u and v are z_i's, they are ordered as $u < v$ by $<'$ which is known to satisfy the safety conditions in S. If u is an x_i and v is a z_j, $u < v$ clearly follows from (3.7).

The remaining case is where $u = z_i$ and $v = x_j$ for some specific z_i and x_j. The safety condition $z_i < x_j$ cannot appear in $W(S,x)$, since it is violated by $<$ in (3.6). It must therefore be the case that $x_j >_S x$. But now $W(S,x)$ will include the safety condition $x_j > x$, in conflict with the ordering given by (3.6). This contradiction completes the proof. ■

Theorem 3.5.7 *Procedure 3.5.5 terminates. The number of calls to* `simp` *is bounded by the size of the problem being solved.*

Proof. In fact, we will not prove the theorem using Procedure 3.5.6 as stated. Instead, consider the following simplification procedure:

Procedure 3.9.1 To compute $\texttt{simp}'(\Gamma, S, \gamma)$:

```
select the conclusion $x$ of $\gamma$ so that $S \cup S_{\{\gamma\}}$ is acyclic
$\Gamma := \Gamma \cup \{\gamma\}$
$S := W(S \cup S_\gamma, x)$
remove from $\Gamma$ any nogood with conclusion $y$ such
        that $y >_S x$
if the conclusions of nogoods in $\Gamma$ rule out all
      possible values for $x$ then
  $\rho :=$ the result of resolving all nogoods in $\Gamma$ with $x$
        in their conclusion
  $\langle \Gamma, S \rangle := \texttt{simp}(\Gamma, S, \rho)$
end if
return $\langle \Gamma, S \rangle$
```

The difference between this procedure and Procedure 3.5.6 is that where Procedure 3.5.6 removed only nogoods with x in their antecedents, Procedure 3.9.1 removes all nogoods with conclusion following x in the partial order $<_S$.

We now have the following:

Lemma 3.9.2 *Suppose that Γ and S are chosen so that $S \supseteq S_\gamma$ for each $\gamma \in \Gamma$. Now if γ is a nogood that violates some acceptable next assignment for Γ and $\langle \Gamma', S' \rangle =$ $\texttt{simp}'(\Gamma, S, \gamma)$, $S' \supseteq S_\gamma$ for each $\gamma \in \Gamma'$.*

Proof. It is clear that the lemma would hold if we were to take $S := S \cup S_\gamma$, so we must only show that the weakening at x cannot drop the safety condition associated with some nogood in Γ'. But if the weakening drops the safety condition $z < y$, it must be that $y >_{S \cup S_\gamma} x$. Since x is the variable in the conclusion of γ, this implies that we must have $y >_S x$, in which case the underlying nogood with y in its conclusion will have been deleted as well. ■

It follows from the lemma that if `simp` removes a nogood γ, `simp'` will drop it as well, since if y is the variable in the conclusion of γ, we clearly have of $y >_{S_\gamma} x$ (since `simp` drops only nogoods with x in their antecedents) so that $y >_S x$ by virtue of the lemma and `simp'` drops the nogood as well.

We therefore see that the difference between the two procedures is only in the set of nogoods maintained; Procedure 3.5.5 as stated retains a superset of the nogoods retained by a version based on $\texttt{simp}'$. The set S of safety conditions is the same in both cases, and the nogood set is acceptable in both cases. It thus suffices to prove the theorem for $\texttt{simp}'$, since the larger set of nogoods computed using $\texttt{simp}$ will simply result in fewer acceptable next assignments for the procedure to consider.

To see that the procedure using $\texttt{simp}'$ terminates, we begin with the following definition:

Definition 3.9.3 *Given a set of safety conditions S and a fixed variable ordering $x_1 < x_2 < \cdots < x_n$ that respects $<_S$, let $m(\Gamma, S, <)$ be the tuple $\langle |x_1|_\Gamma, \ldots, |x_n|_\Gamma \rangle$. We will denote by $\texttt{size}(\Gamma, S, <)$ the size of the remaining search space as given in Definition 3.3.3, and will denote by $\texttt{size}(\Gamma, S)$ the maximum size as $<$ is allowed to vary.*

Proposition 3.9.4 *Suppose that Γ is acceptable, S is acyclic, and $S \supseteq S_\gamma$ for each $\gamma \in \Gamma$. Now if γ is a nogood that violates some acceptable next assignment for Γ and $\langle \Gamma', S' \rangle = \texttt{simp}'(\Gamma, S, \gamma)$, then $\texttt{size}(\Gamma', S') < \texttt{size}(\Gamma, S)$.*

Proof. Let x be the variable in the conclusion of γ. The first nontrivial step of the procedure $\texttt{simp}'$ is $\Gamma := \Gamma \cup \{\gamma\}$. This reduces $|x|_\Gamma$. The next step is $S := W(S \cup S_\gamma, x)$. This introduces new orderings. Let $<$ be any total ordering consistent with $W(S \cup S_\gamma, x)$. There must now exist a total ordering $<'$ which is consistent with $S \cup S_\gamma$ such that $<$ and $<'$ agree through x. Since $|x|_\Gamma$ has been reduced, the tuple associated with $<$ must be lexicographically smaller then the tuple associated with $<'$ at the time the procedure was called. This implies that all tuples allowed after $S := W(S, x) \cup S_\Gamma$ are lexicographically smaller than some tuple allowed at the beginning of the simplification. Applying Lemma 3.3.4, we can conclude that the size of the $\langle \Gamma, S \rangle$ pair has been reduced.

The next step removes from Γ all nogoods with conclusion $y >_S x$. Although this increases the size of the live domain for y, the fact that $y >_S x$ allows us to repeat the lexicographic argument of the preceding paragraph. Finally, if the simplification performs a resolution and executes a recursive call, then that recursion must continue to decrease the size of $\langle \Gamma, S \rangle$. ■

It follows that a modification of Procedure 3.5.5 using $\texttt{simp}'$ will in fact terminate in a number of steps bounded by the original value of $\texttt{size}(\Gamma, S)$, which is the size of the problem being solved. Procedure 3.5.5 itself will terminate no less quickly. ■

Proposition 3.6.5 *Suppose that we are working on a problem with n variables, that the size of the largest domain of any variable is v, and that we have constructed Γ and S using repeated applications of $\texttt{simp}$. If the largest equivalence class $\langle x \rangle_S$ contains d elements, the space required to store Γ is $o(n^2 v^d)$.*

Proof. We know that the nogood set will be acyclic if we group together variables that are equivalent under $\leq_\Gamma$. Since this results in at most d variables being grouped together at any point, the maximum domain size in the reduced problem is v^d and the maximum number of nogoods stored is thus bounded by nv^d. As previously, the amount of space needed to store each nogood is $o(n)$. ■

Theorem 3.6.6 *Procedure 3.6.3 terminates. The number of calls to* `simp` *is bounded by the size of the problem being solved.*

Proof. The proof is essentially unchanged from that of Theorem 3.5.7; we provide only a sketch here. The only novel features of the proof involve showing that the lexicographic size falls as either variables are merged into an equivalence class or an equivalence class is broken so that the variables it contains are once again handled separately. In order to do this, we extend Definition 3.9.3 to handle equivalence classes as follows:

Definition 3.9.5 *Given a set of safety conditions S and a fixed variable ordering $x_1 < x_2 < \cdots < x_n$ that respects $<_S$, let $||x_i||$ be given by*

$$||x_i|| = \begin{cases} 1, & \textit{if } x_i \equiv x_{i+1}; \\ \prod_{y \in \langle x_i \rangle_S} |y|_\Gamma, & \textit{otherwise.} \end{cases} \tag{3.8}$$

Now denote by $\hat{m}(\Gamma, S, <)$ the tuple $\langle ||x_1||, \ldots, ||x_n|| \rangle$. We will denote by $\hat{m}(\Gamma, S)$ that tuple which is lexicographically maximal as $<$ is allowed to vary.

This definition ensures that the lexicographic value decreases whenever we combine variables, since the remaining choices for the combined variable aren't counted until the latest possible point. It remains to show that the removal of nogoods or safety conditions does not split an equivalence class prematurely.

This, however, is clear. If removing a safety condition $y < z$ causes two other variables y_1 and y_2 to become not equivalent, it must be the case that $y_1 \equiv y_2 \equiv z$ before the safety condition was removed. But note that when the safety condition is removed, we must have made progress on a variable $x <_S z$. There is thus no lexicographic harm in splitting z's equivalence class. ■

Experimental code Here is the code used to generate instances of the class of problems on which our ideas were tested. The two arguments to the procedure are the size s of the variable grid and the number c of clauses to be "centered" on any single variable.

For each variable x on the grid we generated either $\lfloor c \rfloor$ or $\lfloor c \rfloor + 1$ clauses at random subject to the constraint that the variables in each clause form a right triangle with horizontal and vertical sides of length 1 and where x is the vertex opposite the hypotenuse. There are four such triangles for a given x. There are eight assignments of values to variable for each triangle giving 32 possible clauses. Our Common Lisp code for generating these 3-SAT problems is given below. Variables at the edge of the grid usually generate fewer than c clauses so the boundary of the grid is relatively unconstrained.

```
(defun make-problem (s c &aux result xx yy)
  (dotimes (x s)
    (dotimes (y s)
      (dotimes (i (+ (floor c)
                     (if (> (random 1.0)
                            (rem c 1.0))
                         0 1)))
        (setq xx (+ x -1 (* 2 (random 2)))
              yy (+ y -1 (* 2 (random 2))))
        (when (and (< -1 xx) (< xx s)
                   (< -1 yy) (< yy s))
          (push (new-clause x y xx yy s)
                result)))))
  result))

(defun new-clause (x y xx yy s)
  (mapcar
   #'(lambda (a b &aux (v (+ 1 (* s a) b)))
       (if (zerop (random 2)) v (- v)))
   (list x xx x) (list y y yy)))
```

Chapter 4

The Hazards of Fancy Backtracking

When dynamic backtracking was tested on some important problem classes, a strange thing happened: performance went *down* relative to traditional methods. Now, it is possible for dependency techniques to degrade performance because the savings in number of nodes examined is outweighed by the extra book-keeping overhead. However, in some notable cases dynamic backtracking was actually increasing the node count. In this paper, presented at AAAI-94 as "The hazards of fancy backtracking" Baker explains how and why this can occur, and offers some suggestions for overcoming the problem.

4.1 Introduction

We are interested in systematic search techniques for solving constraint satisfaction problems. There has been some recent work on intelligent backtracking procedures that can return to the source of a difficulty without erasing the intermediate work. In this paper, we will argue that these procedures have a substantial drawback, but first let us see why they might make sense. Consider an example from [25]. Suppose we are coloring a map of the United States (subject to the usual constraint that only some fixed set of colors may be used, and adjacent states cannot be the same color).

Let us assume that we first color the states along the Mississippi, thus dividing the rest of the problem into two independent parts. We now color some of the western states, then we color some eastern states, and then we return to the west. Assume further that upon our return to the west we immediately get stuck: we find a western state that we cannot color. What do we do?

Ordinary chronological backtracking (depth-first search) would backtrack to the most recent decision, but this would be a state east of the Mississippi and hence irrelevant; the search procedure would only address the real problem after trying every possible coloring for the previous eastern states.

Backjumping [22] is somewhat more intelligent; it would immediately jump back to some state adjacent to the one that we cannot color. In the process of doing this, however, it would erase all the intervening work, i.e., it would uncolor the whole eastern section of the country. This is unfortunate; it means that each time we backjump in this fashion, we will have to start solving the eastern subproblem all over again.

Ginsberg has recently introduced *dynamic backtracking* [25] to address this difficulty. In dynamic backtracking, one moves to the source of the problem without erasing the intermediate work. Of course, simply retaining the *values* of the intervening variables is not enough; if these values turn out to be wrong, we will need to know where we were in the search space so that we can continue the search systematically. In order to do this, dynamic backtracking accumulates nogoods to keep track of portions of the space that have been ruled out.

Taken to an extreme, this would end up being very similar to dependency-directed backtracking [71]. Although dependency-directed backtracking does not save intermediate values, it saves enough dependency information for it to quickly recover its position in the search space. Unfortunately, dependency-directed backtracking saves far too much information. Since it learns a new nogood from every backtrack point, it generally requires an exponential amount of memory — and for each move in the search space, it may have to wade through a great many of these nogoods. Dynamic backtracking, on the other hand, only keeps nogoods that are "relevant" to the current position in the search space. It not only learns new nogoods; it also throws aways those old nogoods that are no longer applicable.

Dynamic backtracking, then, would seem to be a happy medium between backjumping and full dependency-directed backtracking. Furthermore, Ginsberg has presented empirical evidence that dynamic backtracking outperforms backjumping on the problem of solving crossword puzzles [25].

Unfortunately, as we will soon see, dynamic backtracking has problems of its own.

The plan of the paper is as follows. The next section reviews the details of dynamic backtracking. Section 3 describes an experiment comparing the performance of dynamic backtracking with that of depth-first search and backjumping on a problem class that has become somewhat of a standard benchmark. We will see that dynamic backtracking is *worse* by a factor exponential in the size of the problem. Note that this will not be simply the usual complaint that intelligent search schemes often have a lot of overhead. Rather, our complaint will be that the effective search space itself becomes larger; even if dynamic backtracking could be implemented without any additional overhead, it would still be far less efficient than the other algorithms.

Section 4 contains both our analysis of what is going wrong with dynamic backtracking and an experiment consistent with our view. In Section 5, we describe a modification to dynamic backtracking that appears to fix the problem. Concluding remarks are in Section 6.

4.2 Dynamic backtracking

Let us begin by reviewing the definition of a constraint satisfaction problem, or CSP.

Definition 4.2.1 *A constraint satisfaction problem (V, D, C) is defined by a finite set of variables V, a finite set of values D_v for each $v \in V$, and a finite set of constraints C, where each constraint $(W, P) \in C$ consists of a list of variables $W = (w_1, \ldots, w_k) \subseteq V$ and a predicate on these variables $P \subseteq D_{w_1} \times \cdots \times D_{w_k}$. A solution to the problem is a total assignment f of values to variables, such that for each $v \in V$, $f(v) \in D_v$ and for each constraint $((w_1, \ldots, w_k), P)$, $(f(w_1), \ldots, f(w_k)) \in P$.*

Like depth-first search, dynamic backtracking works with partial solutions; a partial solution to a CSP is an assignment of values to some subset of the variables, where the assignment satisfies all of the constraints that apply to this particular subset. The algorithm starts by initializing the partial solution to have an empty domain, and then it gradually extends this solution. As the algorithm proceeds, it will derive new constraints, or "nogoods," that rule out portions of the search space that contain no solutions. Eventually, the algorithm will either derive the empty nogood, proving that the problem is unsolvable, or it will succeed in constructing a total solution that satisfies all of the constraints. We will always write the nogoods in directed form; e.g.,

$$(v_1 = q_1) \wedge \cdots \wedge (v_{k-1} = q_{k-1}) \Rightarrow v_k \neq q_k$$

tells us that variables v_1 through v_k cannot simultaneously have the values q_1 through q_k respectively.

The main innovation of dynamic backtracking (compared to dependency-directed backtracking) is that it only retains nogoods whose left-hand sides are currently true. That is to say that if the above nogood were stored, then v_1 through v_{k-1} would have to have the indicated values (and since the current partial solution has to respect the nogoods as well as the original constraints, v_k would either have some value other than q_k or be unbound). If at some point, one of the left-hand variables were changed, then the nogood would have to be deleted since it would no longer be "relevant." Because of this relevance requirement, it is easy to compute the currently permissible values for any variable. Furthermore, if all of the values for some variable are eliminated by nogoods, then one can resolve these nogoods together to generate a new nogood. For example, assuming that $D_{v_9} = \{1, 2\}$, we could resolve

$$(v_1 = a) \wedge (v_3 = c) \Rightarrow v_9 \neq 1$$

with

$$(v_2 = b) \wedge (v_3 = c) \Rightarrow v_9 \neq 2$$

to obtain

$$(v_1 = a) \wedge (v_2 = b) \Rightarrow v_3 \neq c$$

In order for our partial solution to remain consistent with the nogoods, we would have to simultaneously unbind v_3. This corresponds to backjumping from v_9 to v_3, but without erasing any intermediate work. Note that we had to make a decision about which variable to put on the right-hand side of the new nogood. The rule of dynamic backtracking is that the right-hand variable must always be the one that was most recently assigned a value; this is absolutely crucial, as without this restriction, the algorithm would not be guaranteed to terminate.

The only thing left to mention is how nogoods get acquired in the first place. Before we try to bind a new variable, we will check the consistency of each possible value[1] for this variable with the values of all currently bound variables. If a constraint would be violated, we write the constraint as a directed nogood with the new variable on the right-hand side.

We have now reviewed all the major ideas of dynamic backtracking, so we will give the algorithm below in a somewhat informal style. For the precise mathematical definitions, see [25].

Procedure DYNAMIC-BACKTRACKING

1. Initialize the partial assignment f to have the empty domain, and the set of nogoods Γ to be the empty set. At all times, f will satisfy the nogoods in Γ as well as the original constraints.

2. If f is a total assignment, then return f as the answer. Otherwise, choose an unassigned variable v and for each possible value of this variable that would cause a constraint violation, add the appropriate nogood to Γ.

3. If variable v has some value x that is not ruled out by any nogood, then set $f(v) = x$, and return to step 2.

4. Each value of v violates a nogood. Resolve these nogoods together to generate a new nogood that does not mention v. If it is the empty nogood, then return "unsatisfiable" as the answer. Otherwise, write it with its chronologically most recent variable (say, w) on the right-hand side, add this directed nogood to Γ, and call ERASE-VARIABLE(w). If each value of w now violates a nogood, then set $v = w$ and return to step 4; otherwise, return to step 2.

Procedure ERASE-VARIABLE(w)

1. Remove w from the domain of f.

2. For each nogood $\gamma \in \Gamma$ whose left-hand side mentions w, call DELETE-NOGOOD(γ).

Procedure DELETE-NOGOOD(γ)

[1] A value is possible if it is not eliminated by a nogood.

1. Remove γ from Γ.

Each variable-value pair can have at most one nogood at a given time, so it is easy to see that the algorithm only requires a polynomial amount of memory. In [25], it is proven that dynamic backtracking always terminates with a correct answer.

This is the theory of dynamic backtracking. How well does it do in practice?

4.3 Experiments

To compare dynamic backtracking with depth-first search and backjumping, we will use randomly-generated propositional satisfiability problems, or to be more specific, random 3-SAT problems with n variables and m clauses.[2] Since a SAT problem is just a Boolean CSP, the above discussion applies directly. Each clause will be chosen independently using the uniform distribution over the $\binom{n}{3}2^3$ non-redundant 3-literal clauses. It turns out that the hardest random 3-SAT problems appear to arise at the "crossover point" where the ratio of clauses to variables is such that about half the problems are satisfiable [51]; the best current estimate for the location of this crossover point is at $m = 4.24n + 6.21$ [10]. Several recent authors have used these crossover-point 3-SAT problems to measure the performance of their algorithms [10, 64].

In the dynamic backtracking algorithm, step 2 leaves open the choice of which variable to select next; backtracking and backjumping have similar indeterminacies. We used the following variable-selection heuristics:

1. If there is an unassigned variable with one of its two values currently eliminated by a nogood, then choose that variable.

2. Otherwise, if there is an unassigned variable that appears in a clause in which all the other literals have been assigned false, then choose that variable.

3. Otherwise, choose the unassigned variable that appears in the most binary clauses. A binary clause is a clause in which exactly two literals are unvalued, and all the rest are false.[3]

The first heuristic is just a typical backtracking convention, and in fact is intrinsically part of depth-first search and backjumping. The second heuristic is unit propagation, a standard part of the Davis-Putnam procedure for propositional satisfiability [14, 16]. The last heuristic is also a fairly common SAT heuristic; see for example [10, 80]. These heuristics choose variables that are highly constrained and constraining in an attempt to make the ultimate search space as small as possible.

[2]Each clause in a 3-SAT problem is a disjunction of three literals. A literal is either a propositional variable or its negation.

[3]On the very first iteration in a 3-SAT problem, there will not yet be any binary clauses, so instead choose the variable that appears in the most clauses overall.

Variables	Average Number of Assignments		
	Depth-First Search	Backjumping	Dynamic Backtracking
10	20	20	22
20	54	54	94
30	120	120	643
40	217	216	4,532
50	388	387	31,297
60	709	705	212,596

Table 4.1: A comparison using randomly-generated 3-SAT problems.

For our experiments, we varied the number of variables n from 10 to 60 in increments of 10. For each value of n we generated random crossover-point problems[4] until we had accumulated 100 satisfiable and 100 unsatisfiable instances. We then ran each of the three algorithms on the 200 instances in each problem set. The mean number of times that a variable is assigned a value is displayed in Table 1.

Dynamic backtracking appears to be worse than the other two algorithms by a factor exponential in the size of the problem; this is rather surprising. Because of the lack of structure in these randomly-generated problems, we might not expect dynamic backtracking to be significantly better than the other algorithms, but why would it be *worse*? This question is of more than academic interest. Some real-world search problems may turn out to be similar in some respects to the crossword puzzles on which dynamic backtracking does well, while being similar in other respects to these random 3-SAT problems — and as we can see from Table 1, even a small "random 3-SAT component" will be enough to make dynamic backtracking virtually useless.

4.4 Analysis

To understand what is going wrong with dynamic backtracking, consider the following abstract SAT example:

$$a \rightarrow x \tag{4.1}$$
$$\Rightarrow \neg a \tag{4.2}$$
$$\neg a \Rightarrow b \tag{4.3}$$
$$b \Rightarrow c \tag{4.4}$$
$$c \Rightarrow d \tag{4.5}$$
$$x \Rightarrow \neg d \tag{4.6}$$

Formula (1) represents the clause $\neg a \vee x$; we have written it in the directed form above to suggest how it will be used in our example. The remaining formulas correspond

[4]The numbers of clauses that we used were 49, 91, 133, 176, 218, and 261 respectively.

	Average Number of Assignments		
Variables	Depth-First Search	Backjumping	Dynamic Backtracking
10	77	61	51
20	2,243	750	478
30	53,007	7,210	3,741

Table 4.2: The same comparison as Table 1, but with all variable-selection heuristics disabled.

to groups of clauses; to indicate this, we have written them using the double arrow ($\Rightarrow$). Formula (2) represents some number of clauses that can be used to prove that a is contradictory. Formula (3) represents some set of clauses showing that if a is false, then b must be true; similar remarks apply to the remaining formulas. These formulas will also represent the nogoods that will eventually be learned.

Imagine dynamic backtracking exploring the search space in the order suggested above. First it sets a true, and then it concludes x using unit resolution (and adds a nogood corresponding to (1)). Then after some amount of further search, it finds that a has to be false. So it erases a, adds the nogood (2), and then deletes the nogood (1) since it is no longer "relevant." Note that it does *not* delete the proposition x — the whole point of dynamic backtracking is to preserve this intermediate work.

It will then set a false, and after some more search will learn nogoods (3)–(5), and set b, c and d true. It will then go on to discover that x and d cannot both be true, so it will have to add a new nogood (6) and erase d. The rule, remember, is that the most recently valued variable goes on the right-hand side of the nogood. Nogoods (5) and (6) are resolved together to produce the nogood

$$x \Rightarrow \neg c \tag{4.7}$$

where once again, since c is the most recent variable, it must be the one that is retracted and placed on the right-hand side of the nogood; and when c is retracted, nogood (5) must be deleted also. Continuing in this fashion, dynamic backtracking will derive the nogoods

$$x \Rightarrow \neg b \tag{4.8}$$

$$x \Rightarrow a \tag{4.9}$$

The values of b and a will be erased, and nogoods (4) and (3) will be deleted.

Finally, (2) and (9) will be resolved together producing

$$\Rightarrow \neg x \tag{4.10}$$

The value of x will be erased, nogoods (6)–(9) will be deleted, and the search procedure will then go on to rediscover (3)–(5) all over again.

	Average Number of Assignments		
Variables	Depth-First Search	Backjumping	Dynamic Backtracking
10	20	20	20
20	54	54	53
30	120	120	118
40	217	216	209
50	388	387	375
60	709	705	672

Table 4.3: The same comparison as Table 1, but with dynamic backtracking modified to undo unit propagation when it backtracks.

By contrast, backtracking and backjumping would erase x before (or at the same time as) erasing a. They could then proceed to solve the rest of the problem without being encumbered by this leftover inference. It might help to think of this in terms of search trees even though dynamic backtracking is not really searching a tree. By failing to retract x, dynamic backtracking is in a sense choosing to "branch" on x before branching on a through d. This virtually doubles the size of the ultimate search space.

This example has been a bit involved, and so far it has only demonstrated that it is *possible* for dynamic backtracking to be worse than the simpler methods; why would it be worse in the *average* case? The answer lies in the heuristics that are being used to guide the search.

At each stage, a good search algorithm will try to select the variable that will make the remaining search space as small as possible. The appropriate choice will depend heavily on the values of previous variables. Unit propagation, as in equation (1), is an obvious example: if a is true, then we should immediately set x true as well; but if a is false, then there is no longer any particular reason to branch on x. After a is unset, our variable-selection heuristic would most likely choose to branch on a variable other than x; branching on x anyway is tantamount to randomly corrupting this heuristic. Now, dynamic backtracking does not really "branch" on variables since it has the ability to jump around in the search space. As we have seen, however, the decision not to erase x amounts to the same thing. In short, the leftover work that dynamic backtracking tries so hard to preserve often does more harm than good because it perpetuates decisions whose heuristic justifications have expired.

This analysis suggests that if we were to eliminate the heuristics, then dynamic backtracking would no longer be defeated by the other search methods. Table 2 contains the results of such an experiment. It is important to note that all of the previously listed heuristics (including unit resolution!) were disabled for the purpose of this experiment; at each stage, we simply chose the first unbound variable (using some fixed ordering). For each value of n listed, we used the same 200 random problems that were generated earlier.

The results in Table 2 are as expected. All of the algorithms fare far worse than before, but at least dynamic backtracking is not worse than the others. In fact, it is a bit better than backjumping and substantially better than backtracking. So given that there is nothing intrinsically wrong with dynamic backtracking, the challenge is to modify it in order to reduce or eliminate its negative interaction with our search heuristics.

4.5 Solution

We have to balance two considerations. When backtracking, we would like to preserve as much nontrivial work as possible. On the other hand, we do not want to leave a lot of "junk" lying around whose main effect is to degrade the effectiveness of the heuristics. In general, it is not obvious how to strike the appropriate balance. For the propositional case, however, there is a simple modification that seems to help, namely, undoing unit propagation when backtracking.

We will need the following definition:

Definition 4.5.1 *Let v be a variable (in a Boolean* CSP*) that is currently assigned a value. A nogood whose conclusion eliminates the other value for v will be said to* justify *this assignment.*

If a value is justified by a nogood, and this nogood is deleted at some point, then the value should be erased as well. Selecting the given value was once a good heuristic decision, but now that its justification has been deleted, the value would probably just get in the way. Therefore, we will rewrite DELETE-NOGOOD as follows, and leave the rest of dynamic backtracking intact:

Procedure DELETE-NOGOOD(γ)

```
1. Remove $\gamma$ from $\Gamma$.
2. For each variable $w$ justified by $\gamma$, call ERASE-VARIABLE($w$).
```

Note that ERASE-VARIABLE calls DELETE-NOGOOD in turn; the two procedures are mutually recursive. This corresponds to the possibility of undoing a cascade of unit resolutions. Like Ginsberg's original algorithm, this modified version is sound and complete, uses only polynomial space, and can solve the the union of several independent problems in time proportional to the sum of that required for the original problems.

We ran this modified procedure on the same experiments as before, and the results are in Table 3. Happily, dynamic backtracking no longer blows up the search space. It does not do much good either, but there may well be other examples for which this modified version of dynamic backtracking is the method of choice.

How will this apply to non-Boolean problems? First of all, for non-Boolean CSPs, the problem is not quite as dire. Suppose a variable has twenty possible values, all but two of which are eliminated by nogoods. Suppose further that on this basis, one of the remaining values is assigned to the variable. If one of the eighteen nogoods is later eliminated, then the variable will still have but three possibilities and will probably remain a good choice. It is only in the Boolean problems that an assignment can go all the way from being totally justified to totally unjustified with the deletion of a single nogood. Nonetheless, in experiments by Jónsson and Ginsberg it was found that dynamic backtracking often did worse than depth-first search when coloring random graphs [38]. Perhaps some variant of our new method would help on these problems. One idea would be to delete a value if it loses a certain number (or percentage) of the nogoods that once supported it.

4.6 Discussion

Although we have presented this research in terms of Ginsberg's dynamic backtracking algorithm, the implications are much broader. Any systematic search algorithm that learns and forgets nogoods as it moves laterally through a search space will have to address—in some way or another—the problem that we have discussed. The fundamental problem is that when a decision is retracted, there may be subsequent decisions whose justifications are thereby undercut. While there is no *logical* reason to retract these decisions as well, there may be good heuristic reasons for doing so.

On the other hand, the solution that we have presented is not the only one possible, and it is probably not the best one either. Instead of erasing a variable that has lost its heuristic justification, it would be better to keep the value around, but in the event of a contradiction remember to backtrack on this variable instead of a later one. With standard dynamic backtracking, however, we do not have this option; we always have to backtrack on the most recent variable in the new nogood. Ginsberg and McAllester have recently developed *partial-order dynamic backtracking* [30], a variant of dynamic backtracking that relaxes this restriction to some extent, and it might be interesting to explore some of the possibilities that this more general method makes possible.

Perhaps the main purpose of this paper is to sound a note of caution with regard to the new search algorithms. Ginsberg claims in one of his theorems that dynamic backtracking "can be expected to expand fewer nodes than backjumping provided that the goal nodes are distributed randomly in the search space" [25]. In the presence of search heuristics, this is false. For example, the goal nodes in *unsatisfiable* 3-SAT problems are certainly randomly distributed (since there are not any goal nodes), and yet standard dynamic backtracking can take orders of magnitude longer to search the space.

Therefore, while there are some obvious benefits to the new backtracking techniques, the reader should be aware that there are also some hazards.

Chapter 5

The Application of Satisfiability Algorithms to Scheduling Problems

In this chapter we turn from randomly generated problems to more structured job-shop scheduling problems. The results are rather surprising: a simple probing algorithm called ISAMP outperforms both state-of-the art local search and backtracking techniques.

This work appeared as "Experimental results on the application of satisfiability algorithms to scheduling problems" by Crawford and Baker, at AAAI-94.

5.1 Introduction

Many classes of problems in knowledge representation, learning, planning, and other areas of AI are known to be NP-complete. In the worst case, all known algorithms for solving such problems require run time exponential in the size of the problem. Propositional satisfiability (SAT) is, in a sense, the prototypical example of an NP-complete problem. It is simply formalized, yet has an amazingly complex structure.

Paradoxically, one perennial problem with work on SAT has been the difficulty of finding hard instances on which to test algorithms; it turns out to be surprisingly hard to collect a sufficiently large body of reasonably sized "real" problems. Randomly generated problems, on the other hand, tend to end up being quite easily solved.

One important advance in recent years has been the discovery that the difficulty of randomly generated problems depends critically on whether they are *under-constrained*, *over-constrained*, or *critically-constrained* [5, 52, 11]. Consider a randomly generated constraint satisfaction problem. Intuitively, if there are very few constraints, it should be easy to find a solution (since there will generally be many solutions). Similarly, if there are very many constraints then an intelligent algorithm will generally be able to quickly close off most or all of the branches in the search tree. The hardest problems are thus those which are critically constrained: these problems have relatively few solutions, but most branches in the search tree go fairly deep before reaching a dead end.

Critically constrained randomly generated satisfiability problems provide a ready supply of hard test cases of arbitrary size. This discovery has lead to work on understanding [11, 77], and solving [63] these problems. GSAT in particular appears to be well suited to solving large randomly generated critically constrained problems. However, concerns have been raised that randomly generated problems are bad test cases because they have no structure and thus may bear little resemblance to "real" problems.

This paper reports on a series of experiments applying satisfiability algorithms to scheduling problems. In these experiments we have used Sadeh's job shop scheduling problems (both because they are readily accessible and because they have been well studied in the scheduling community). These problems do have a random component, namely the ready times and deadlines for the jobs. However, these times have been chosen according to a variety of distributions in an effort to mimic various types of scheduling problems encountered in the field.

Our main result from these experiments is that Sadeh's scheduling problems bear little resemblance to critically constrained 3SAT, but not for the expected reason. When translated into SAT problems, these scheduling problems are much larger than previously studied random 3SAT problems. However, they are still solvable because they are highly under-constrained – very many solutions exist so it is a fairly simple matter to "bump" into one. However, GSAT has not proven to be the best algorithm on these problems. We hypothesize that this is due to the presence of a small number of "control" variables (those define the schedule) and a much larger number of "dependent" variables (whose values are determined by the control variables). Since GSAT has no notion of forward checking, it appears to have considerable difficulty with problems involving large numbers of dependent variables (this effect is discussed further in in the discussion section).

TABLEAU, a Davis-Putnam derivative, also performed poorly. On some problems it almost immediately found a solution. On others, however, it made an initial bad guess and was stuck searching a virtually infinite search tree (typically these trees were of depth seventy to eighty which means that the search trees have on the order of 2^{70} nodes). We refer to this as the *early mistake problem.*

This mode of failure suggested replacing depth-first search (in TABLEAU) with iterative sampling [45]. Iterative sampling is a simple technique in which variable values are chosen at random (but with forward checking[1]) until a model or a contradiction is found. At this point we return to the *root* of the search tree and start over. Iterative sampling successfully solved all of Sadeh's scheduling problems after an average of only 64 restarts (using no heuristics). This confirms that these problems have a very large number of solutions, and suggests that the domain-specific heuristics commonly used in scheduling problems are less useful than might be expected.

[1]Adding forward checking to iterative sampling seems to have been first suggested by Kurt Konolige.

If these problems are truly representative of "real" constraint-satisfaction problems, these results suggest quite a different research agenda than has previously been pursued. Large under-constrained constraint-satisfaction problems pose a number of interesting challenges not found in critically constrained problems. Chief among these is the early mistake problem. Forward checking also seems to be important, so GSAT is not necessarily the solution. The best current candidates seem to be hill-climbing algorithms with forward checking (see discussion section) and variants of dynamic backtracking [25]. It is also becoming clear that a purely propositional representation is impractical – a large fraction of the run time is spent simply reading the theory in from disk. Generalizing existing SAT algorithms to use some sort of "semi-first-order" shorthand is clearly in order. One other important question that currently remains open is whether these scheduling problems are perhaps similar to *under-constrained* 3SAT problems. This seems relatively unlikely (since forward checking appears to be more important in scheduling than in any of the randomly generated problems) but deserves to be investigated.

5.2 Scheduling

The scheduling problem is ubiquitous. One may, for example, have a set of machine tools and be told to schedule a series of jobs so as to maximize the efficiency of the use of the tools. Alternatively, one may have a collection of transport ships in various locations and be told to transport some number of divisions to a variety of locations as reliably and cheaply as possibly. Or, one may have to assemble some number of surgical teams using a variety of specialists subject to a set of constraints on consecutive numbers of hours worked, availability of operating rooms, etc. In all cases, the general form of the problem is that one is given a set of tasks to achieve, and a collection of resources to use.

One important type of scheduling problems is *machine shop scheduling* [78, 68]. Sadeh has developed a test suite of machine shop scheduling problems that are intended to represent a range of the types of machine shop scheduling problems encountered in the field.

Machine shop scheduling problems are usually taken to consist of a number of operations $1, \ldots, n$ to be scheduled subject to a collection of constraints. Each operation requires processing time p_i (given as part of the problem). A solution is a schedule giving the start time, s_i, for each operation.

The constraints are usually taken to consist of *sequencing restrictions*, *resource capacity constraints*, and *ready times and deadlines* [68]. Sequencing restrictions, written $i \longrightarrow j$ state that operation i must complete before j can begin. The restriction $i \longrightarrow j$ is thus equivalent to $s_i + p_i \leq s_j$ ("the start time of i plus processing time for i is less than or equal to the start time of j"). Resource capacity constraints, written $c_{i,j}$, state that operations i and j conflict (usually because both require the same resource) and thus cannot be scheduled concurrently. $c_{i,j}$ is equivalent to the disjunction $(s_i + p_i \leq s_j) \vee (s_j + p_j \leq s_i)$ ("i completes before j begins or j completes before i begins"). Ready times, represented by r_i, are the earliest time at which operation i can start. A deadline d_i is the time by which operation i must be completed. Ready times thus just state $s_i \geq r_i$ and deadlines that $s_i + p_i \leq d_i$.

We now discuss propositional satisfiability (SAT) and then show how scheduling problems can be converted into SAT problems.

5.3 Propositional Satisfiability

The propositional satisfiability problem is the following [20]: Given a set of *clauses*[2] C on a finite set U of variables, find a truth assignment[3] for U that satisfies all the clauses in C.

Clearly one can determine whether a satisfying assignment exists by trying all possible assignments. Unfortunately, if the set U is of size n then there are 2^n such assignments. SAT algorithms thus typically either (1) walk through the space of assignments following some set of heuristics and hope to run into a solution, or (2) work with partial assignments and use some sort of *forward checking* to compute forced values for other variables. Algorithms in this second class generally use depth-first search to systematically search the space of assignments. In section on satisfiability algorithms below we discuss examples of both types of algorithms.

5.4 Encoding Scheduling Problems as SAT Problems

At first glance there seems to be an "obvious" translation of scheduling problems into SAT: create variables to represent the start times of the operations (*e.g.*, s_{it} true means operation s_i starts at time t), and create clauses to represent the necessary inequalities. However, the search space in SAT problems so generated turns out to be much larger than necessary.

To see this, note that the key decisions to be made in scheduling problems concern the *orderings* of conflicting operations. Thus, for example, if i and j share a resource then we have to decide whether to schedule i then j, or j then i. We do not necessarily have to specify the exact start times of operations i and j, as long as we can be sure that there is some way to do so that is consistent with our ordering decisions (this observation, and the essence of the encoding we use here, is due to Smith and Cheng (1993)).

More formally, for each pair of operations i and j, we introduce a boolean variable $pr_{i,j}$ meaning "i precedes j", and for each operation i and each time t we introduce

[2]A clause is a disjunction of variables or negated variables.

[3]A truth assignment is a mapping from U to $\{true, false\}$.

a boolean variable $sa_{i,t}$ meaning "i starts at time t or later", and a boolean variable $eb_{i,t}$ meaning "i ends by time t."[4]

Scheduling constraints are then translate by:

$i \longrightarrow j$ becomes $pr_{i,j} = true$
$c_{i,j}$ becomes $pr_{i,j} \vee pr_{j,i}$
r_i becomes $sa_{i,r_i} = true$
d_i becomes $eb_{i,d_i} = true$

We refer to the set of constraints generated by this mapping as $\mathcal{C}$.

It is also necessary to add a collection of "coherence conditions" on the introduced variables. In all conditions below, i and j are quantified over all relevant operations and t is quantified over all relevant times.

1. $sa_{i,t} \rightarrow sa_{i,t-1}$ (coherence of sa). This ensures that if i starts at or after time t then it starts at or after time $t-1$.

2. $eb_{i,t} \rightarrow eb_{i,t+1}$ (coherence of eb). This ensures that if i ends by t then it ends by $t+1$.

3. $sa_{i,t} \rightarrow \neg eb_{i,t+p_i-1}$ (job i requires time p_i). This ensures that if i starts at or after time t then it cannot end before time $t + p_i$.

4. $sa_{i,t} \wedge pr_{i,j} \rightarrow sa_{j,t+p_i}$ (coherence of $pr_{i,j}$). This ensures that if i start at or after t and j follows i then j cannot start until i is finished.

We refer to the set of coherence conditions as $\mathcal{S}$. Any mapping of the variables $p_{i,j}$ to $\{T, F\}$ that extends to a model of $\mathcal{C} \wedge \mathcal{S}$ is then a template describing a set of solutions to the scheduling problem.

There are several advantages to this translation from scheduling problems to SAT:

1. A set of legal values for the pr variables corresponds to a collection of feasible schedules. If there are additional optimization conditions (*e.g.*, robustness), one can then use these to select a particular schedule.

2. The constraints in $\mathcal{S}$ apply to all scheduling problems and thus need be computed and stored only once (they might therefore be compiled into a procedure rather than being stored explicitly). Further, $\mathcal{S}$ is symmetric under any permutation of the operations. In order to check for symmetries among operations it is thus only necessary to consider $\mathcal{C}$.

[4]To help avoid confusion, in this section all boolean variables (variables taking values $\{true, false\}$) are of length two (*e.g.*, $pr_{i,j}$) and variables taking integral values (*e.g.*, s_i) are of length one.

3. Without loss of generality, one can omit the variable $pr_{i,j}$ (and all clauses containing it) if there are no sequencing restrictions or resource capacity constraints for i and j. This means that the size of the SAT problem is of order $nd + c$ (where n is the number of operations, d is the number of distinct time points, and c is the number of constraints in the scheduling problem). If $\mathcal{S}$ is represented as a compiled procedure then the number of clauses in the SAT problem is further reduced to order c (plus the size of the compiled procedure).

5.5 Three Satisfiability Algorithms

5.5.1 Tableau

TABLEAU is a Davis-Putnam algorithm that does a depth-first search of possible assignments using unit propagation for forward checking. This basic algorithm dates back to the work of Davis, Logemann, and Loveland [13]. TABLEAU adds efficient data-structures for fast unit propagation and a series of heuristics for selecting branch variables (these are discussed in [11]). Most of these heuristics do not seem particularly helpful for scheduling problems (see discussion of experimental results below).

Unit propagation consists of the repeated application of the inference rule:

$$\frac{\begin{array}{l} x \\ \neg x \vee y_1 \ldots \vee y_n \end{array}}{y_1 \vee \ldots \vee y_n}$$

(similarly for $\neg x$). Unit propagation is a special case of resolution (the singleton x is resolved against the $\neg x$ in the clause). It is a particularly useful case, however, since it can always be performed to completion in time linear in the size of the theory, and since, in practice, it greatly reduces the number of nodes in the search tree (by propagating variable values through the theory).

The basic depth-first search algorithm is then the following:

```
tableau(theory)
  unit_propagate(theory);
  if contradiction discovered return(false);
  else if all variables are valued
            return(current assignment);
  else {
    x = some unvalued variable;
    return(tableau(theory AND x) OR
           tableau(theory AND NOT x));
  }
```

5.5.2 Gsat

GSAT is the most successful hill-climbing search algorithm for SAT to date. A complete assignment of variables to values is always kept. Variables are "flipped" (their value is changed) so as to increase the number of satisfied clauses (if possible). In our experiments we found the use of the "walk" strategy [61] to be critical. The basic WSAT ("walk sat") algorithm is the following:

```
WSAT(theory)
  for i := 1 to MAX-TRIES {
    A := a randomly generated truth assignment;
    for j := 1 to MAX-FLIPS {
      if A is a solution return it;
      else {
        C := randomly chosen unsatisfied clause;
        With probability P,
          Flip a random variable in C;
        Otherwise (that is, with probability 1-P)
          Flip a variable in C resulting in the
           greatest decrease in the number of
                   unsat clauses;
      }
    }
  }
  return failure
}
```

The experimental performance of GSAT [63] with walk on certain problem classes is impressive. GSAT is often able to find models for randomly generated 2000 variable critically-constrained 3SAT problems.[5] Systematic methods (methods that are guaranteed to always to find a solution or determine that none exists) are currently not able to solve any critically-constrained problems of this size.

5.5.3 Isamp

ISAMP is basically a variant of TABLEAU in which one gives up on backtracking and simply starts over whenever a contradiction is discovered:

```
Isamp(theory) {
  for i := 1 to MAX-TRIES {
    set all variables to unassigned;
    loop {
      if all variables are valued
                return(current assignment);
      v := random unvalued variable;
      assign v a randomly chosen value;
      unit_propagate();
      if contradiction exit loop;
    }
  }
  return failure
}
```

[5]Since GSAT never determines unsatisfiability, there is no way to reliably determine what percentage of satisfiable problems GSAT solves (but it is believed to be high).

Obviously this approach will only work on problems with a large number of models. One of the surprises in our work on scheduling problems has been that this algorithm outperforms both TABLEAU and GSAT.

5.6 Experimental Results

Our experiments were designed to assess the performance of each of these three algorithms on scheduling problems. We used the sixty scheduling problems produced by Sadeh [57]. Each of these problems consists of fifty operations to be scheduled subject to sequencing restrictions and resource capacity constraints. The operations are grouped into ten jobs of five operations each. Operations within each job must be performed in order. Further, each job requires one of five resources and each resource can be used by at most one job at a time.

Ready times and deadlines were generated randomly using several distributions. The distributions were defined by two parameters: (1) degree of constraint: (w) wide, (n) narrow, and (t) tight, and (2) number of bottlenecks: none, one, or two. These two parameters yield the six classes shown in Figure 5.1. Sadeh produced ten sample problems from each class.

The results for ISAMP and GSAT are shown in figure 5.1. The GSAT results are for our version of the WSAT ("walk sat") variant of GSAT [61]. These results are an average over ten runs on each problem. In each run GSAT was given ten tries of four million flips each which corresponds to about forty-five minutes of computation time. The mean flip and time data is for the successful cases only (these counts would be higher if we "punished" GSAT for the cases on which it ran out of time). For ISAMP the results are an average over 100 runs. ISAMP runs were given twenty-thousand tries. TABLEAU was run with only the non-horn heuristic ("branch first on variables appearing in non-horn clauses"). This has the effect of forcing TABLEAU to branch on the *pr* variables.[6] Since TABLEAU currently has no random component, the results are for one run on each problem. TABLEAU was interrupted after forty-five minutes. The averages are over the successful cases only. All algorithms are implemented in C, and all experiments were run on a SPARC 10/51.

In order to test the hypothesis that the lack of unit propagation was hurting GSAT, we performed a linear time simplification on the propositionally encoded scheduling problems and ran the experiments again. The simplification consisted of running unit propagation to completion on the initial theory (the encoding is such that the initial theories contain a number of unit clauses). We then deleted any clauses that were subsumed by a unit literal (*e.g.*, if the theory contained x and $x \vee y \vee z$ then we deleted the clause). Finally we "compacted" the encoding – our encoding uses integers to represent variables so this step ensured that if the largest variable in the

[6]We have some evidence to suggest that branching on randomly chosen variables would be better for these problems, but have not yet finished this experiment with TABLEAU.

TABLEAU:

Class	Success Rate	Branches	Time (sec.)
w/1	90	3947.1	255.4
w/2	100	221.2	104.8
n/1	70	1719.7	79.2
n/2	100	93.8	90.6
t/1	80	1160.4	66.3
t/2	100	119.4	81.7

GSAT:

Class	Success Rate	Flips (millions)	Time (sec.)
w/1	99	6.7	500
w/2	99	7.0	599
n/1	100	3.2	258
n/2	100	3.7	312
t/1	96	3.9	274
t/2	88	5.0	354

ISAMP:

Class	Success Rate	Tries	Time (sec.)
w/1	100	7	10
w/2	100	15	13
n/1	100	13	11
n/2	100	45	21
t/1	100	52	19
t/2	100	252	68

Figure 5.1: Experimental results for scheduling problems.

theory is n then every integer less than n is also used for some variable in the theory. The data for the simplified theories is shown in Figure 5.2. As expected, the GSAT run times are much lower. The ISAMP run times are also lower. We believe that this is primarily due to the fact that the simplified theory is smaller (and thus faster to read in from disk). We did not rerun TABLEAU since the simplification should have minimal effect on its run time. In this experiment GSAT and ISAMP were each run ten times on each problem.

5.7 Discussion

The success of ISAMP indicates that, given the right encoding, Sadeh's scheduling problems are not that difficult. The encoding we use is domain specific only in that it is implicitly based on the observation that the key choices to be made are the relative

GSAT:

Class	Success Rate	Flips (millions)	Time (sec.)
w/1	100	0.39	27
w/2	100	0.29	23
n/1	100	0.31	23
n/2	100	0.55	43
t/1	100	1.1	77
t/2	97	2.9	211

ISAMP:

Class	Success Rate	Tries	Time (sec.)
w/1	100	7	7
w/2	100	18	10
n/1	100	13	8
n/2	100	42	15
t/1	100	62	16
t/2	100	180	43

Figure 5.2: Experimental results for scheduling problems after simplification.

orders of the conflicting operations. This is much less domain-specific than the slack-based heuristics used by Smith and Cheng [68]. Of course Smith and Cheng do solve these problems almost two orders of magnitude faster than ISAMP . However, most of this difference is probably due to our use of propositional logic as a representation language (just to read in a propositional version of these theories takes about thirty times as long as Smith and Cheng take to solve them). The obvious experiment of implementing ISAMP using a more natural representation language (*e.g.*, a constraint-satisfaction language with integral valued variables) is underway.

A great deal of recent work has been done on analysis and solution methods for randomly generated satisfiability problems. An important open question in this body of work has been its relevance to "real" problems. Our work attempts to begin providing an answer to this question by studying the performance of a variety of satisfiability algorithms on propositional encodings of scheduling problems.

We have found that scheduling problems generate propositional theories that are much larger and much less constrained than the randomly generated theories generally studied. Further, the variables in scheduling problems can be partitioned into *control* variables that define a solution (*e.g.*, the *pr* variables in scheduling problems) and *dependent* variables whose values are derived from the control variables (*e.g.*, all the rest of the variables in the scheduling problems).

The fact that TABLEAU generally performs poorly on these problems while ISAMP performs well indicates that there are a large number of solutions but that these solutions are not uniformly distributed throughout the search space. Rather there seem to be large "deserts" containing no solutions. TABLEAU sometimes wanders into one of these deserts by making an unlucky choice at some early branch in the tree. It then has no way to recover. We refer to this as the early mistake problem. Since ISAMP restarts on every contradiction, it is sensitive only to the number of solutions, not their uniformity.

The existence of a large number of dependent variables appears to be hobbling GSAT relative to ISAMP . One can see this by the following analysis. Assume that we can divide the variables in a problem into c control variables and d dependent variables, such that a polynomial time procedure will always determine whether an assignment to the control variables extends to a model. This will be the case, for example, if we choose the control variables to be variables appearing in non-horn clauses (*e.g.*, the *pr* variables in scheduling problems) and choose unit propagation as the polynomial time procedure. If there are *any* constraints on the dependent variables the density of solutions in "control variable space" will then be higher than density of solutions in the original search space (to see this note that the density could be equal only if any satisfying assignment to the control variables extended to 2^d models). GSAT clearly searches in the full space (since it uses no forward checking). We hypothesize that ISAMP effectively searches in a smaller space (even though it makes no explicit distinction between control and dependent variables[7]) because of its use of unit propagation (TABLEAU also used unit propagation but it appears to fail because of the early mistake problem discussed above). Work on GSAT suggests that hill-climbing is a useful technique for satisfiability problems and in general one would expect hill-climbing to be superior to the random probing of ISAMP . This clearly suggests that the next step in this line of work is to develop algorithms that hill-climb in control space.

[7]When we do explicitly force ISAMP to value only the *pr* variables its performance *falls*. As yet we have no explanation for this phenomenon.

Chapter 6

Limited Discrepancy Search

The reason that ISAMP works for scheduling problems is that value propagation is critical – once some decisions have been made many other decisions are forced. This means that local search, which has no notion of value propagation, is ineffective. Backtracking search does utilize value propagation but it fails because of the *early mistake problem* – decisions made early in the search cannot be undone until huge portions of the search space are traversed. On even small job-shop scheduling problems this is often fatal.

Limited discrepancy search (LDS) was one of the most important innovations to come out of this project. LDS uses value propagation, but avoids the early mistake problem. It turns out to be a quite general method that is applicable in almost any domain where the search space is huge, but where we have available a good, but imperfect, heuristic.

This work first appeared as "Limited discrepancy search" by Harvey and Ginsberg, in IJCAI-95.

6.1 Introduction

In practice, many search problems have spaces that are too large to search exhaustively. One can often find solutions while searching only a small fraction of the space by relying on carefully tuned heuristics to guide the search toward regions of the space that are likely to contain solutions. For many problems, heuristics can lead directly to a solution—most of the time. In this paper, we consider what to do when the heuristics fail.

We will focus our attention on procedures for tree search. Our objective is simple: For search problems with heuristically ordered successors, we will develop a search procedure that is more likely to find a solution in any given time limit than existing methods such as chronological backtracking and iterative sampling [46]. The outline of this paper is as follows: In the next section, we discuss existing algorithms. Limited discrepancy search (LDS) is introduced in Section 6.3 and compared to existing techniques in Section 6.4. We discuss variations of LDS that we believe will be useful for solving realistic problems in Section 6.5. We conclude by presenting our experimental results in Section 6.6.

6.2 Existing Strategies

Consider a tree search problem for which the successor ordering heuristic is so good that it almost always leads directly to a solution. Such problems are common both in practice and in areas of AI research such as planning and scheduling [67, 76]. If the heuristic is good enough, one might be satisfied with an algorithm that follows the heuristic and just gives up if the heuristic fails to lead to a solution, an algorithm we will call "1-samp" [35, 67]. If the performance of 1-samp is not satisfactory, however, one is confronted with the question of what search algorithm to use instead. Iterative sampling and backtracking are two candidates.

6.2.1 Iterative Sampling

Iterative sampling [46], or *isamp*, is the simple idea of following random paths, or probes, from the root until eventually a path that leads to a solution is discovered. At each node on a path, one of the successors is selected at random and expanded. Then one of its successors is selected at random, and so on until a goal node or dead end is reached. If the path ends at a dead end, isamp starts a new probe, beginning again at the root.

Since the algorithm samples with replacement, there is a uniform chance of finding a goal node on any particular probe. Provided that there is a goal node somewhere in the space, it follows that the probability of find a goal node increases uniformly toward 1 as the number of probes grows without limit.

Iterative sampling has been shown to be effective on problems where the solution density is high [9], but its performance as a fallback procedure for 1-samp is questionable because it ignores the successor-ordering heuristic. If the heuristic were the key to solving the problem despite a low solution density, one would not expect iterative sampling to be effective.[1]

6.2.2 Backtracking

An alternative fallback procedure is simply to backtrack chronologically when 1-samp fails. Our experiments in Section 6.6 with scheduling show that this approach provides little improvement over 1-samp itself, and the analysis of mistakes provides an explanation [36]. There is a reasonable chance that, somewhere early in 1-samp's

[1] We have experimented with biasing the random selection of successors according to the heuristic, but our results suggest this is not a viable approach [36].

path, it made a mistake by selecting a successor that had no goal nodes in the entire subtree below it. Once this early mistake is made and the successor's subtree is committed to, none of the subsequent decisions makes any difference.

If the subtree below a mistake is large, chronological backtracking will spend all of the allowed run time exploring the empty subtree, without ever returning to the last decision that actually matters. If one is counting on the heuristics to find a goal node in a small fraction of the search space, then chronological backtracking puts a tremendous burden on the heuristics early in the search and a relatively light burden on the heuristics deep in the search. Unfortunately, for many problems the heuristics are *least* reliable early in the search, before making decisions that reduce the problem to a size for which the heuristics become reliable. Because of the uneven reliance on the heuristics, it is unlikely that chronological backtracking is making the best use of the heuristic information.

6.3 Discrepancies

Let us return to the search problems for which the successor ordering heuristic is a good one. Our intuition is that, when 1-samp fails, the heuristic probably would have led to a solution if only it had not made one or two "wrong turns" that got it off track. It ought to be possible to systematically follow the heuristic at all but one decision point. If that fails, we can follow the heuristic at all but two decision points. If the number of wrong turns is small, we will find a solution fairly quickly using this approach.

We call the decision points at which we do not follow the heuristic "discrepancies." *Limited discrepancy search* embodies the idea of iteratively searching the space with a limit on the number of discrepancies allowed on any path. The first iteration, with a limit of zero discrepancies, is just like 1-samp. The next iteration searches all possibilities with at most one discrepancy, and so on.

The algorithm is shown in Figure 6.1. We will assume the search tree is binary. SUCCESSORS is a function that returns a list of the either zero or two successors, with the heuristic preference first.

In Figure 6.1, x is the discrepancy limit. We iteratively call LDS-PROBE, increasing x each time. LDS-PROBE does a depth-first search traversal of the tree, limiting the number of discrepancies to x. When eventually x reaches d, the maximum depth of the tree, LDS-PROBE searches the entire tree exhaustively. Thus the search is guaranteed to find a goal node if one exists and is guaranteed to terminate if there are no goal nodes.

Since each iteration of LDS-PROBE limits the number of discrepancies to x instead of restricting the search to those nodes with exactly x discrepancies, iteration n reexamines the nodes considered by previous iterations (see Figure 6.2). As with other iterative techniques, however, the final iteration is far and away the most expensive

```
LDS-PROBE(node, k)
    if GOAL-P(node) return node
    $s \leftarrow$ SUCCESSORS(node)
    if NULL-P($s$) return NIL
    if $k = 0$ return LDS-PROBE(FIRST($s$), 0)
    else
        result $\leftarrow$ LDS-PROBE(SECOND($s$), $k - 1$)
        if result $\neq$ NIL return result
        return LDS-PROBE(FIRST($s$), $k$)
```

```
LDS(node)
    for $x \leftarrow 0$ to maximum depth
        result $\leftarrow$ LDS-PROBE(node, $x$)
        if result $\neq$ NIL return result
    return NIL
```

Figure 6.1: Limited Discrepancy Search.

and the redundancy is therefore not a significant factor in the complexity of the search.

Figure 6.2 shows a trace of LDS exhaustively searching a full binary tree of height three. The heuristic orders nodes left to right. The twenty pictures show all the paths to depth three, in order. The dotted lines and open circles represent nodes that were not backtracked over since the previous picture, so the trace can be followed by examining at the pictures in sequence. Counting all the black circles gives the total number of nodes expanded in the search, forty.

In general, the number of nodes expanded by LDS with a discrepancy limit of x is bounded by d^{x+1} (for iteration x, there are at most d^x fringe nodes, with each path to a fringe node expanding at most d nodes). If d is large, the cost of any single iteration dominates the summed costs of the preceding ones.

6.4 Comparison with existing methods

In practice, of course, we will typically not have time to search the space exhaustively. We would therefore like to know the likelihood of finding a solution, using the various methods, in the amount of time we are actually willing to wait. We will make this question precise by formalizing what we mean for the heuristic to make a "wrong turn."

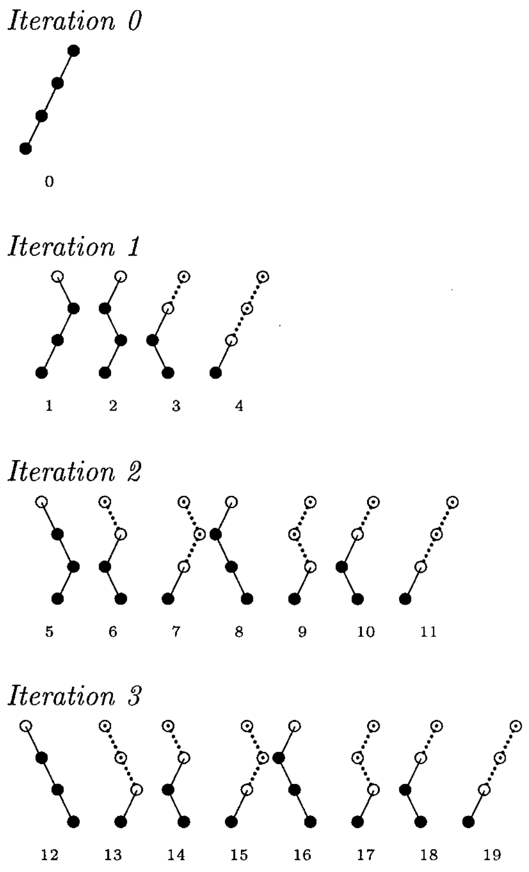

Figure 6.2: Execution trace of LDS.

6.4.1 Wrong Turns

For simplicity, we will consider only the case of a full binary tree. The two children of each choice point are assumed to be in the order of heuristic preference. We will further assume that if a choice point has a goal node in the subtree below it, then with probability p (the *heuristic probability*) its first child has a goal node in its subtree. If the first child does not have a goal, the other child must have a goal since the choice point has only two children. In this case the heuristic has made a *wrong turn* by putting the children in the "wrong" order.

The notion of a wrong turn is closely related to the mistake probability. We define a *bad* node to be a node that does not have any goal nodes in its subtree. We define a *mistake* to be a bad node whose parent is not bad. The *mistake probability*, m, is the probability that a randomly selected child of a good node is bad [36]. If the heuristic orders successors randomly, the heuristic probability is the complement of the mistake probability, $p = 1 - m$. If the heuristic does better than random selection, $p > 1 - m$.

Figure 6.3 shows the four possibilities for a node and its children. An $\times$ indicates

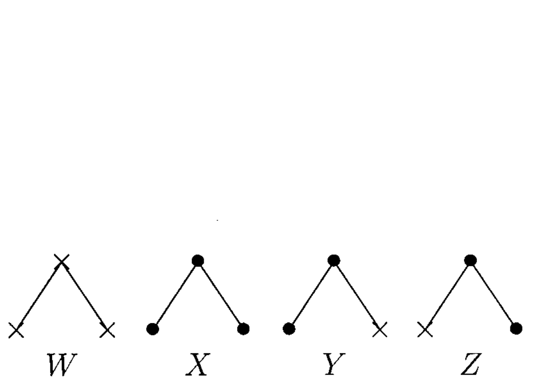

Figure 6.3: The four possibilities for a node and its children.

a bad node, a solid dot a good node. In the figure, p is the probability that a node is in class X or Y (the two classes with good left children) given that it is not in class W (the only class where the parent is bad). The mistake probability m is one half the probability that a node is in class Y or Z (the classes with one mistake child) given that it is not in class W.

Experimentally, it appears that m is generally fairly constant throughout many search trees [36]. In order to simplify our analyses, we will assume that p is constant as well, although the experimental evidence is that p tends to increase somewhat as we search the tree because most heuristics are more accurate at deep nodes than at shallow ones.

The chance of finding a solution on a random path to depth d (i.e., using isamp) is simply $(1-m)^d$. Using heuristics and assuming a constant p, 1-samp has probability p^d of finding a solution on its one and only path.

This observation allows us to estimate p by running 1-samp on a large training set of problems from the domain of interest. Let s be the success rate of 1-samp on the training set. Since the probability of success for 1-samp is p^d, we have $p = s^{1/d}$. If s is small, the training set may have to be impractically large to get a reliable estimate. For some problems, though, s is not small. Heuristics developed for job shop scheduling have been shown to yield a probability s that is nearly one for small research problems [67]. We have found in earlier experimental work on the same problems [35] that even standard CSP heuristics can yield a success rate of about 75%. On larger scheduling problems [73] the success rate of 1-samp is less, but more sophisticated heuristics from operations research keep 1-samp competitive with other search techniques [6].

6.4.2 Theoretical results

Given specific values for m and for p, Figures 6.4 and 6.5 show the theoretical probability of success as a function of time for iterative sampling (isamp), chronological backtracking (DFS), and limited discrepancy search (LDS) for various heuristic probabilities p.[2] The graphs show the probability of finding a solution in some number of probes i, where we define a probe to be a search until a dead end is reached for isamp

[2]The combinatoric manipulations underlying these figures are quite involved and appear elsewhere [36].

or LDS, or simply a search of an additional d nodes for DFS. The number of probes is limited by the height of the tree because the combinatorics of solving the problem beyond the one-discrepancy limit are intractable. The analyses are biased toward DFS because depth-first search is given the highest of the heuristic probabilities shown in each figure.

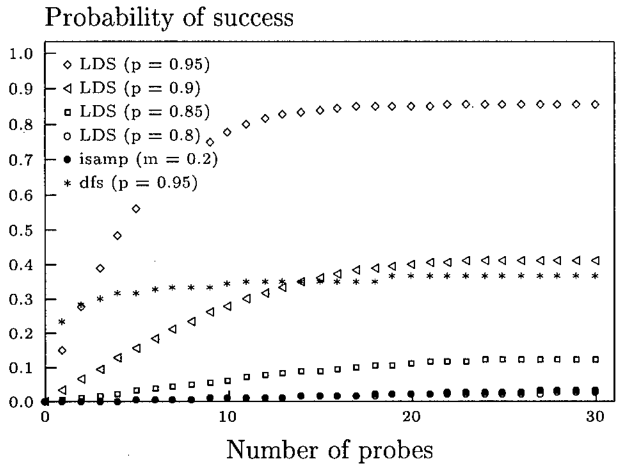

Figure 6.4: A problem of height 30.

Figure 6.4 shows results for a problem of height 30, with a mistake probability m of 0.2. The problem has about a billion fringe nodes of which a few more than a million are goals.[3] With a solution density of 1/1000, we would expect iterative sampling to sample about 500 fringe nodes before finding a solution (807, to be exact).[4] By many accounts, a problem with a solution density of 0.001 is a fairly easy problem. It takes only $807 \cdot 30 = 24,210$ nodes, on average, to find a solution using iterative sampling. The expected number of probes is slightly more than the number of probes required to have a 50% chance of finding a solution, $560 \cdot 30 = 16,800$.[5]

In practice, we may be interested in the number of nodes required to find a solution with a higher probability of success. The number of nodes required by iterative sampling for a success probability of 0.8 on this problem is $1300 \cdot 30 = 39,000$. Compare this to the performance of limited discrepancy search. For $p = 0.95$, LDS has probability of success 0.8 with just eleven probes, or 990 nodes. The savings, nearly a factor of forty, depends on the heuristic to order successors correctly seven out of eight times when one of the successors is a mistake.

[3]The number of goals is $(2 - 2m)^d$.

[4]The expected number of probes is $1/(1 - m)^d$.

[5]For example, the expected number of coin flips before getting heads is two, whereas it takes only a single coin flip to have probability 0.5 of getting heads. The number of probes required to achieve a given success probability s is $\frac{\log(1-s)}{\log(1-(1-m)^d)}$.

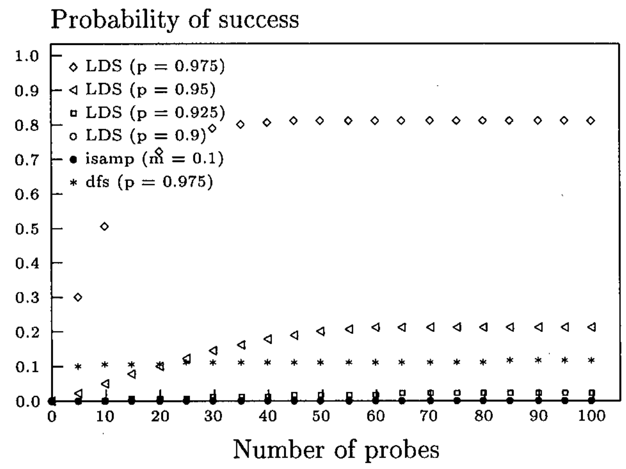

Figure 6.5: A problem of height 100.

For $p = 0.8 = 1 - m$, the heuristic orders the successors correctly half the time, no better than random selection. The $p = 0.8$ curve in the figure (almost completely obscured by the isamp curve) shows that the performance of LDS is slightly worse than iterative sampling under these conditions. For $p = 0.85, 0.9$, and 0.95 the heuristic orders nodes correctly five, six, and seven out of eight times. The curves show that the expected performance of LDS increases dramatically with the better p.

The DFS curve for $p = 0.95$ rises only marginally above the probability 0.21 that its first fringe node is a goal.[6] The futility of DFS is even clearer in the deeper search shown in Figure 6.5.

The problem of Figure 6.5 has height 100, and approximately 10^{30} nodes. The density of solutions for $m = 0.1$ is about 2.6×10^{-5}. Iterative sampling needs 26,096 probes (2.6 million nodes) to have a 50% chance of success. If, as in the earlier problem, the heuristic orders nodes correctly seven out of eight times ($p = 0.975$ for $m = 0.1$), LDS has a similar chance with just twenty probes (2,000 nodes), a savings of three orders of magnitude over iterative sampling. The savings is similar if a success probability of 0.8 is desired instead.

For higher probabilities of success, the three orders of magnitude savings is more doubtful, though perhaps not as doubtful as the graph seems to suggest. The one discrepancy iteration ends after 101 probes. The later probes of the one discrepancy iteration have much of their paths in common, so the likelihood that one of these later probes succeeds given that the others failed is small. After 101 probes, though, the two discrepancy iteration begins to explore "fresh" paths again. Consequently, we would expect the LDS curve to rise steadily once again where the graph leaves off.[7]

[6]The probability that the first fringe node examined by DFS is a goal is simply p^d.

[7]As remarked earlier, we are unfortunately unable to verify this essentially theoretical claim because the combinatorics overwhelm us.

6.5 Variations and Extensions

The reason we have focused on analyzing the early iterations of limited discrepancy search is that we believe in practice they are the only iterations that matter. Earlier, we argued on intuitive grounds that they would be more important than the later iterations. We will now take the position that in practice the later iterations don't matter at all. The reason is that if the objective is to maximize the probability of finding a solution in a given number of nodes, there are always better things to do than use those nodes on later iterations of limited discrepancy search.

This section discusses a few of the more promising choices. Since some involve combinations with other techniques and others depend on search space properties that are difficult to quantify, this discussion will be less precise than that of earlier sections. Here, we will view limited discrepancy search as a tool that can be used in combination with other techniques to craft an effective search procedure for a given real world problem.

6.5.1 Variable Reordering

Constraint satisfaction problems and SAT problems are formulated as tree search by fixing an order for the variables to be instantiated or determining the order dynamically as the search progresses. In either case, a node in the search tree is a choice point for the possible instantiations of a particular variable. If an effective heuristic does not solve the problem with a limit of one discrepancy for some chosen variable order, it may still solve the problem with one discrepancy given a different variable order. The "wrong turn" instantiations that the heuristic makes on the first variable order may even follow from unit propagation on the second. This suggests the simple technique of repeating the one discrepancy iteration of LDS with different variable orders. When variable order is determined dynamically, it may suffice simply to begin the search with a different variable on each iteration. A similar technique improves the efficiency of depth-first search as well (see Section 6.6).

6.5.2 Using Different Heuristics

If multiple heuristics exist for a particular problem, one can try repeating the one- or two-discrepancy limit iterations of LDS with different heuristics. If one heuristic is unlucky and makes more than two wrong turns on a given problem, some other heuristic may be luckier. In general, what is hard for one heuristic may not be hard for another. LDS is an effective way to give one heuristic a reasonable chance before switching to another.

6.5.3 Combining LDS with Bounded Backtrack Search

LDS can also be combined with bounded backtrack search (BBS) [36] to produce an algorithm that does not count "small" discrepancies (those that fail quickly) toward the discrepancy limit. This algorithm can also be viewed as modifying the heuristic to avoid choices that can be seen to fail using a fixed lookahead. (The algorithm itself appears in the last section of this chapter.)

The combined LDS-BBS algorithm outperforms both LDS and BBS on job shop scheduling problems. In fact, LDS-BBS appears to be the algorithm of choice among all systematic backtracking strategies in this domain. There is a compelling theoretical argument for this. Many mistakes result in quick, if not immediate, failures. If a heuristic makes few wrong turns to begin with, it makes even fewer wrong turns that exceed the backtrack bound. Adding a bounded backtrack enables limited discrepancy search to discover solutions with a discrepancy limit of no more than the number of wrong turns that actually exceed the backtrack bound, potentially reducing the number of required iterations. Since the cost of each LDS iteration grows by a factor of d, the savings can be substantial. The added cost of the backtrack bound is relatively insignificant. Adding a backtrack bound of one node can cost at most a factor of 2. A backtrack bound of l costs at most a factor 2^l and, for small l, is likely to be cheaper than the cost of an additional iteration. This upper bound is conservative since the heuristic, by assumption, makes few mistakes.

6.5.4 Local Optimization Using LDS

For problems like scheduling, LDS can also be used to search the neighborhood of an existing solution. The one discrepancy iteration of LDS is modified to begin following the path of the previous best solution instead of following the heuristic. At the depth of the discrepancy, the algorithm diverges from the previous solution and follows the heuristic for the remaining decisions. If the path ends in a solution that is better than the previous best, it can be adopted immediately or stored as a contender for the basis of the next iteration.

This variation of LDS requires some measure of the "goodness" of a solution. For scheduling problems, the schedule length is often the appropriate measure. Searching for a schedule that takes less than time L, if successful, produces a schedule that takes time L'. A set of standard LDS iterations can be repeated with the lower time bound L', or the optimization variant of LDS can be applied to consider variations of the previous schedule that differ by at most one discrepancy.[8]

[8]Alternatively, the time bound can be adjusted by binary division. A single iteration of LDS, though, is not a decision procedure, so failure to find a schedule for a given time bound is no proof that no such schedule exists.

6.5.5 Non-Boolean problems

Finally, we should at least comment on the possible extension of LDS to constraint-satisfaction problems involving variables with domain sizes larger than 2. Although we have focussed on Boolean problems in this paper (in part because the most natural encoding of job-shop scheduling problems is Boolean [67]), the technique can obviously be applied in a wider setting. There are a variety of choices that will need to be made, however: Should the one-discrepancy search include every alternate value for the variable that violates the heuristic, or only the single next most attractive choice? If the number of nodes expanded is to increase by a factor of no more than d on each iteration, we will need to take the latter view.

6.6 Experimental Results

Our experimental results comparing limited discrepancy search with chronological backtracking and iterative sampling are based on a set of thirteen job shop scheduling problems taken from a recent survey of operations research techniques [73].[9] Each of the problems involves scheduling the tasks that might be involved in producing widgets in a manufacturing setting: Each job j_i needs to be performed on a particular machine m_i and takes time t_i. There are constaints indicating that some jobs need to be completed before others can be started, and so on.

The most effective encoding of problems such as these focusses directly on the resource contentions that arise; if two jobs j_i and j_k require the same machine, we introduce a variable p_{ik} indicating whether it is job j_i or job j_k that uses the machine first [67]. Because these variables are Boolean, the search space is far smaller than it would be if we were to make the variables the start times of the various jobs themselves.

Our experimental work formulated each problem as a CSP with a loose bound on the schedule length. We then iteratively repeated the search, decreasing the bound each time to slightly less than the length of the last schedule found. We recorded the length of the best schedule found as a function of the total number of nodes expanded until reaching a final cutoff of 500,000 nodes per problem (see Figure 6.6).

At any given node cutoff $M \leq 500,000$, each algorithm had completed some number of iterations for each problem, resulting in schedules of various lengths. We evaluated the schedules by these lengths, measuring their percent above the optimal length for each problem.[10] We took the average percent above optimal (a function of M) as the overall measure of the performance of the search algorithms.

In Figure 6.6, LDS is clearly superior to chronological backtracking and iterative sampling. We chose this particular benchmark, though, so that we could also compare

[9]The problems can be obtained by sending a message to `o.rlibrary@ic.ac.uk`.

[10]The optimal lengths were taken to be the best reported lengths as of November, 1994.

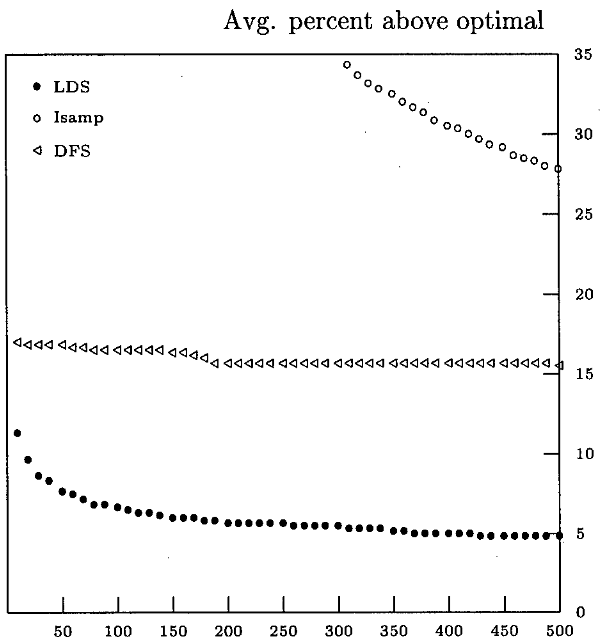

Figure 6.6: Comparison to DFS and iterative sampling.

our results with other scheduling research in artificial intelligence and operations research. On this benchmark, contemporary OR scheduling programs score in the range 0.45% to 8.31% above optimal [73]. Although the performance of our implementation does not match the best of these programs, it appears to be in the same range.

Our scheduling implementation uses general CSP heuristics, which are weak by scheduling standards. Relative to the larger pool of programs, our implementation appears to be comparable using limited discrepancy search but disastrous using chronological backtracking and iterative sampling. Since limited discrepancy search relies heavily on the heuristics, we expect that the combination of LDS with the more accurate heuristics of the dedicated scheduling programs would have the best performance overall. Experiments in this vein are under way.

We also experimented with a variety of nonsystematic algorithms [36]. Depth-first search with restarts, iterative broadening, and bounded backtrack search scored 4.9%, 4.6%, and 4.2% above optimal on the benchmark and all outperformed pure limited discrepancy search slightly (LDS was also 4.9% above optimal).[11]

However, since all of these nonsystematic methods rely less on the heuristics than LDS, we believe that LDS is likely to benefit more significantly from future improve-

[11]Although the overall difference of 0.7% between the best and worst of these algorithms may appear slight, it is substantial in this domain, since the problems can be expected to become exponentially more difficult as one approaches the crossover point corresponding to optimality [8].

ments to the heuristics. As we commented in Section 6.5.3, limited discrepancy search can also be combined with bounded backtrack search. The results are shown in Figure 6.7.[12]

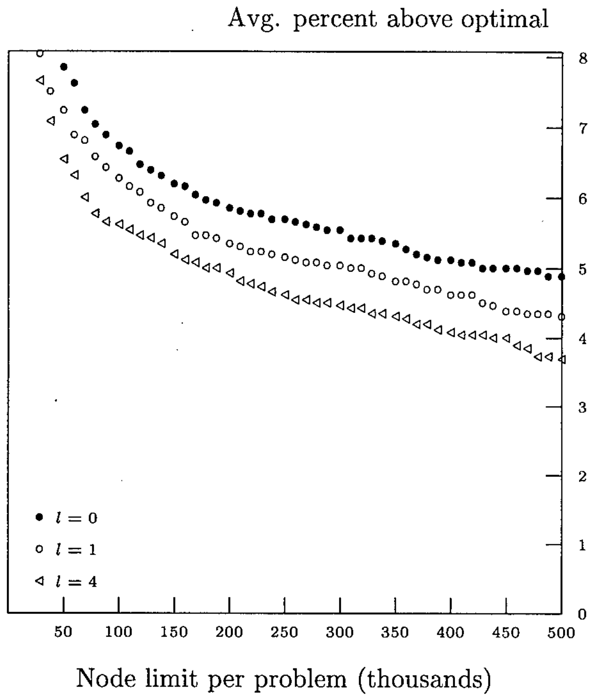

Figure 6.7: Adding bounded backtrack improves BBS.

The combination of limited discrepancy with bounded backtrack search had the best performance of all the systematic and nonsystematic methods we tested. At 3.68% over optimal, its performance with a four-node backtrack bound is respectable when compared to the dedicated scheduling programs.

6.7 Discussion

We have shown both theoretically and experimentally that limited discrepancy search is an effective way to exploit heuristic information in tree search problems. It is more effective than either chronological backtracking or iterative sampling, and we have attempted to explain why.

The scheduling problems that we used in our experiments, while large by contemporary research standards, are not large relative to the types and sizes of scheduling problems it would be useful to solve in the real world. Because of the complexity of

[12]The parameter l in the figure is the depth of backtrack allowed in checking for heuristics that led to dead ends.

scheduling, it is likely that the challenge of scaling up from research problems to real world problems will be met more quickly by advances in heuristics than by the evolution of brute force methods. We expect that in the future, techniques that depend on heuristics yet recover gracefully by searching alternatives when the heuristics fail will be the methods of choice for solving real world problems.

6.8 The combined LDS-BBS algorithm

```
LB-PROBE(node, k, look)
1  if GOAL-P(node) return ⟨node, 0⟩
2  s ← SUCCESSORS(node)
3  if $k > 0$, s ← REVERSE(s)
4  $i \leftarrow 0$
5  count ← 0
6  maxheight ← 0
7  for child in s
8      if $k = 0$ and $count \geq 1$ break
9      if $k \geq 0$ and $i = 0$ $k' \leftarrow k - 1$
10     else $k' \leftarrow k$
11     ⟨result, height⟩ ← LB-PROBE(child, k', look)
12     maxheight ← MAX(maxheight, 1 + height)
13     if result $\neq$ NIL return ⟨result, 0⟩
14     $i \leftarrow i + 1$
15     if $height \geq look$, count ← count + 1
16 return ⟨ NIL, maxheight ⟩
```

```
LDS-BBS(node, look)
1  for $x \leftarrow 0$ to maximum depth
2      ⟨result, height⟩ ← LB-PROBE(node, x, look)
3      if result $\neq$ NIL return result
4  return NIL
```

Chapter 7

Scaling Up

The real question, of course, is not whether one can solve academic benchmarks, but whether one can solve scheduling problems of realistic size and character. In this chapter we turn to a body of work aimed at doing just that. The results to date have been quite good: as of this writing the CIRL scheduler is able to find the best known solutions to the set of large resource-constraints project scheduling problems posted to the internet by Barry Fox of McDonnell Douglas and Mark Ringer of Honeywell.

The work described here was first reported on in "An approach to resource constrained project scheduling" by Crawford appearing in the proceedings of the *1996 Artificial Intelligence and Manufacturing Research Planning Workshop*, published by AAAI press.

7.1 Introduction

Historically there has often been a mismatch between the types of scheduling problems that have been studied academically and the needs of the manufacturing community. The most obvious difference is that many real problems are much larger than common academic benchmarks. A second, and equally important, difference is that real problems generally involve constraints that have a more complex structure than can be expressed within a limited framework like job shop scheduling.

In this paper we overview ongoing work on resource constrained project scheduling (RCPS). RCPS is a generalization of job shop scheduling in which tasks can use multiple resources, and resources can have a capacity greater than one. RCPS is thus a good model for problems, like aircraft assembly, that cannot be expressed as job shop problems. In fact, if we take arguably the most widely used commercial scheduling program, Microsoft Project, RCPS seems to capture exactly the optimization problem that the "resource leveler" in Project solves.

It turns out that the algorithms that have been developed for job shop scheduling do not work particularly well for RCPS. In this paper we overview an approach to RCPS that is based on the combination of limited discrepancy search (LDS) with a novel optimization technique. The resulting system produces the best known schedules for problems of realistic size and character.

7.2 Resource Constrained Project Scheduling

A Resource Constrained Project Scheduling (RCPS) problem consists of a set of tasks, and a set of finite capacity resources. Each task puts some demand on the resources. For example, changing the oil might require one workman and one car lift. A partial ordering on the tasks is also given specifying that some tasks must preceded others (*e.g.*, you have to sand the board before you can paint it). Generally the goal is to minimize makespan without violating the precedence constraints or over-utilizing the resources.

RCPS is more general than job shop because resources can have capacity greater than one, and because tasks can use a collection of resources. This allows resources to be taken to be anything from scarce tools, to specialized workmen, to work zones (such as the cockpit of an airplane). RCPS problems arise in applications ranging from aircraft assembly to chemical refining.

Formally, Resource Constrained Project Scheduling is the following:

Given: A set of tasks T, a set of resources R, a capacity function $C : R \rightarrow \mathbb{N}$, a duration function $D : T \rightarrow N$, a utilization function $U : T \times R \rightarrow \mathbb{N}$, a partial order P on T, and a deadline d.

Find: An assignment of start times $S : T \rightarrow \mathbb{N}$, satisfying the following:

1. Precedence constraints: if t_1 precedes t_2 in the partial order P, then $S(t_1) + D(t_1) \leq S(t_2)$.

2. Resource constraints: For any time x, let $running(x) = \{t | S(t) \leq x < S(t) + D(t)\}$ Then for all times x, and all $r \in R$, $\sum_{t \in running(x)} U(t,r) \leq C(r)$.

3. Deadline: For all tasks t : $S(t) \geq 0$ and $S(t) + D(t) < d$.

7.3 A Benchmark Problem

Our experiments have been run on a series of problems made available on the WWW at:

```
http://www.neosoft.com/~benchmrx
```

by Barry Fox of McDonnell Douglas and Mark Ringer of Honeywell, serving as Benchmarks Secretary in the AAAI SIGMAN and in the AIAA AITC, respectively. These problems have 575 tasks and 17 resources. Some of the resources represent zone (geometric) constraints, and some represent labor constraints. Labor availability varies by shift. This is a synthetic problem that has been generated from experience with multiple large scale assembly problems. It is comparable to real problems is size and character, but simpler in the complexity of the constraints.

The results that have been posted to the WWW to date are shown in figure 7.1. Mark Ringer's results were found using a simple, first fit, interval based algorithm with no optimization. They were posted to encourage other contributions, rather than to generate the best solutions possible.

Who	Problem			Note
	2	3	4	
Mark Ringer	45	57	56	Honeywell
Nitin Agarwal	40	47	57	SAS
Colin Bell	39	-	-	Univ. of Iowa
Barry Fox	38	45	42	McDonnell Douglas
Crawford *et. al.*	38	43	41	CIRL
Crawford *et. al.*	38	39	38	(Lower Bound)

Figure 7.1: Results

7.4 Solution Methods

The two most important methods used in our scheduler are doubleback optimization and limited-discrepancy search. We discuss each in turn and then discuss how they work together in the scheduler.

7.4.1 Doubleback Optimization

The optimizer starts with any schedule satisfying the precedence constraints and generates a legal (and often a shorter) schedule. It works in two steps: a right shift and then a left shift (a very similar technique, *schedule packing* was independently invented previously by Barry Fox [1996]).

We first establish the right hand end point. Recall that the availability of some labor resources varies by shift. Because of this it turns out that it matters where in the daily cycle the endpoint is set.[1] The best heuristic seems to be to set it at the same point in the daily cycle as the end point of the current schedule. Another approach is to select the right hand end point randomly. This tends to shake things up a bit and makes iterating the optimizer more effective.

Once the right hand endpoint is selected we right shift the schedule. In the right shift we take the tasks in order of decreasing finish times (i.e., from the right hand end). We shift each task as far right as it will go. In doing this shift we consider

[1]This was first observed by Joe Pemberton.

only precedence constraints and resource constraints with previously shifted tasks (one way to think about this is to envision the new right hand endpoint as being at positive infinity: the tasks that have not yet been shifted do not interfere with the construction of the new schedule).

Once the right shift is completed, we left shift back to time zero, starting from the beginning of the right shifted schedule. As before, we left shift as far as possible subject to precedence constraints, and resource conflicts with previously left shifted tasks.

This sequence can be iterated. At some points this produces longer schedules (possibly then followed by shorter schedules after additional iterations). At present we have no theoretical method to predict the optimal number of iterations: we simply iterate ten times and keep the best schedule produced.

To see the optimization works, consider the example shown in figure 7.2.

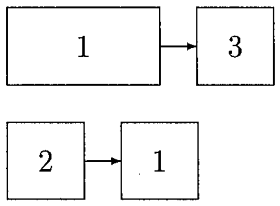

Figure 7.2: A simple scheduling problem.

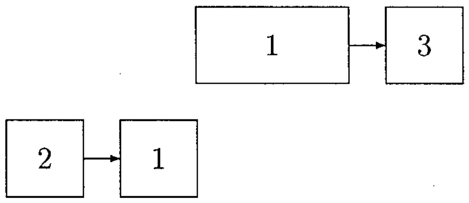

Figure 7.3: A bad schedule.

Here the boxes represent tasks and the arrows represent precedence relations. The numbers in the boxes are the resources the tasks need. For this example all resources have capacity one.

In order to break the resource conflict between the two tasks using resource one, we have to establish an ordering between the tasks. Assume that we do this non-optimally, generating the schedule shown in figure 7.3.[2]

Now consider what happens when we apply the optimizer. The right-shifted, and then left-shifted, schedules are shown in figure 7.4. The key thing to notice is that

[2]Such a mistake is unlikely in such a small problem, but our ability to avoid analogous mistakes in larger problems is, in a sense, the entire source of the intractability of scheduling.

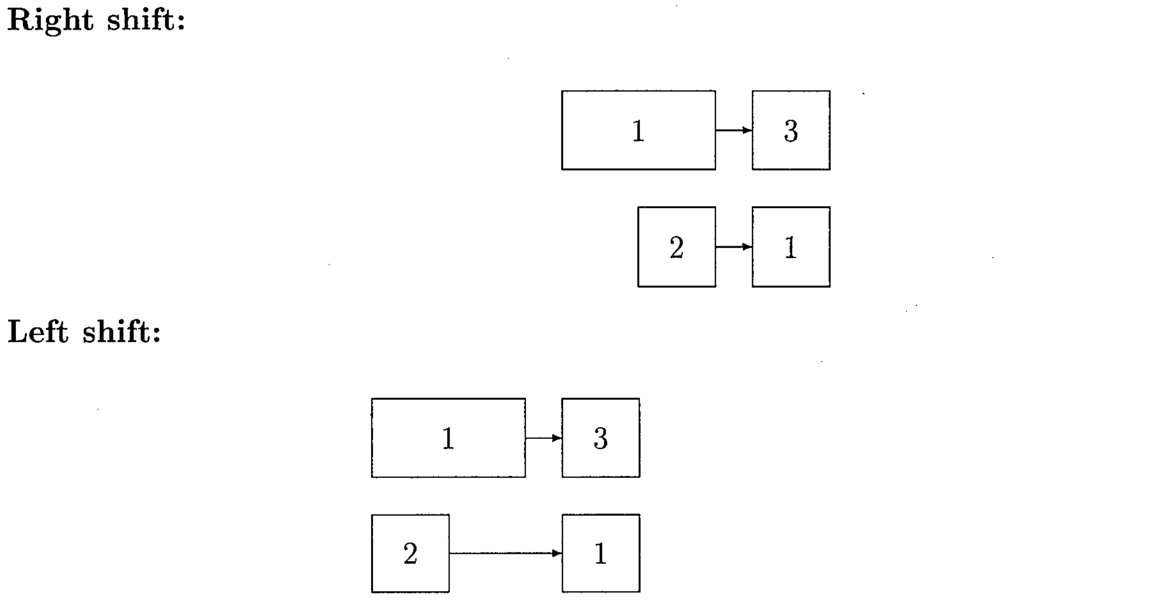

Figure 7.4: The effect of a right and then left shift.

in the right shift the bottom task falls to the end of the schedule (because it has no successors), while the top task is forced to the beginning of the schedule. Thus the left shift schedules the top task first, generating the optimal schedule.

We can, of course, generate test cases in which the optimizer fails to find the shortest schedule, and we currently cannot offer any theoretical guarantees on the optimizer's performance. The strongest statement we can make is that on the benchmark examples, the optimizer is experimentally the single most effective scheduling technique we are aware of.

7.5 Why the Optimizer Works

Experimentally doubleback optimization is quite effective. Starting with the schedule that starts each task as early as possible subject to only the precedence constraints, the optimizer is able to produce 43 day schedules for problem 4. The obvious question is why such a simple technique works so well.

In a sense the key decision to be made in scheduling is the ordering of tasks that compete for a resource. In fact the search portion of our approach (see below) essentially searches all possible ways to break resource contention by establishing an ordering between competing tasks.

The challenge of breaking resource contention is that we do not know which task will turn out to be the most important. In some cases it is obvious which tasks are most critical. For example, if a task must be followed by a long series of tasks, then clearly we want to give it a high priority – otherwise it will be postponed and the "tail" of tasks that follow it is likely to exceed the deadline. Unfortunately it is not generally this simple (or else scheduling would be tractable). In essence what goes wrong is that we do not know how hard it will be to schedule the sets of tasks that must follow the conflicting tasks. However, if we start from a "seed" schedule, then we can decide with reference to the seed, how hard the subsequent tasks will be, and use this information to make better decisions about task priorities.

It turns out that this is exactly what the optimizer does. The right shift pushes all tasks as late as possible. So, if a task is near the beginning of the right shifted schedule, it is there because it must be followed by a large number of tasks. So it should be given high priority. This is exactly what the left shift does: the left shifted schedule is formed by first schedule the tasks that are near the beginning of the right shifted schedule.

7.6 Limited Discrepancy Search

The results returned by the optimizer are sensitive to the "seed" schedule given to the optimizer. One can construct examples of "bad" schedules that the optimizer cannot correct. In a sense the optimizer is walking down to a kind of local minimum, and the quality of the final schedule depends on where the walk starts.

Our implementation uses LDS [37] to produce a series of seed schedules that are then passed to the optimizer. Here we give a brief overview of LDS. Details can be found in Harvey and Ginsberg [1995].

Imagine that we have a schedule that satisfies the precedence constraints, but not the resource constraints. The natural way to produce a legal schedule is to iteratively pick a resource conflict, delay one or more tasks long enough to break the conflict, and then propagate these delays through the precedence constraints. This is, in fact, how our current implementation works (starting from the left shifted schedule satisfying only the precedence constraints).

Each conflict that is broken creates a choice point, and breaking a series of conflicts produces a search tree. In the case of the benchmark problems this produces a search tree with a branching factor of about three, and a depth of about 1000.

The traditional approach to searching such a tree is to use a depth-first search. In a depth-first search, we make a series of decisions until we reach a leaf node in the tree (in this case a leaf node is a schedule satisfying both precedence and resource constraints). We then back up to the last choice point and take the other branch, and follow it to a leaf node. We then back up again, this time to the latest branch point that still has unexplored children. Repeating this we eventually search the entire tree, and are thus guaranteed to find the optimal schedule.

Unfortunately, if the search tree is 1000 nodes deep then a depth-first search will examine only a tiny fraction of the entire search tree (backing up perhaps 10 or 20 nodes). Further, it is reasonable to expect that the choices that will be reconsidered are exactly the choice for which the heuristic is most likely to have made the right decision. To see why this is so, notice that near the top of the search tree there are still many resource conflicts, so the heuristics are working from a "schedule" that is far from legal, so the heuristics are having to guess at how the resolution of these other conflicts will interact with the current conflict. Near the bottom of the tree, however, the schedule is in nearly its final form so the heuristics have good information on which to base their decisions.

As a result, traditional depth-first search is relatively little help on scheduling problems. This has lead many practitioners to either use no search (just following the heuristic and returning the first schedule produced) or to use a local search (which can reconsider any decision at any point).

In LDS we fix a bound on the number of times we will diverge from the heuristic. If that bound is zero then we just produce the single schedule given by always following the heuristics. If the bound is one then we produce a set of schedules generated by ignoring the heuristic exactly once.

The difference between LDS and depth-first search is illustrated in figures 7.5 and 7.6. In both search trees the branch preferred by the heuristic is always drawn on the left. In figure 7.5 the leaves are numbered according to the order in which depth-first search will visit them. Notice that if the heuristic makes a mistake high in the tree, for example, at the first choice point, then depth-first search will have to search half of the search tree before correcting the mistake. In the LDS search tree (figure 7.6), leaf nodes are labeled according to how many times the path from the root to the leaf diverges from the heuristic (*i.e.*, how many right turns are necessary to reach the leaf). LDS searches the leaves by first searching the leaf marked 0, then all the leaves marked 1, and so on.

If the heuristic is generally correct, but sometimes makes mistakes (as if generally the case with heuristics) then we can reasonably expect to find good quality schedules by searching the nodes for which the number of divergences is low. If the height of the tree is h, then the complexity of visiting each node with d divergences is h^d. In practice we usually set d to 1 or 2. This produces much better results than a depth-first search examining the same number of nodes. Further, unlike a local search, LDS is systematic: if we continue to raise d we are guaranteed to eventually find the optimal schedule.

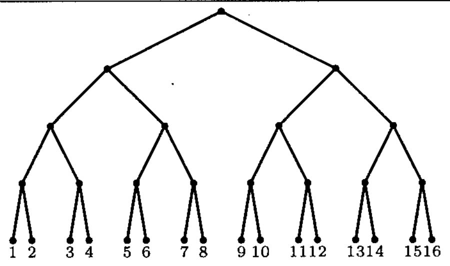

Figure 7.5: Backtracking search tree.

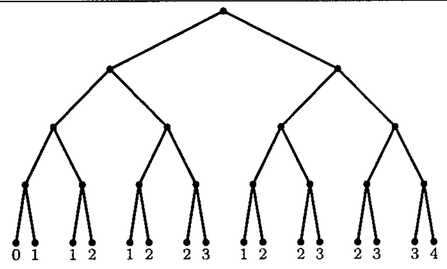

Figure 7.6: LDS search tree.

Finally we should node that LDS and the optimizer work well together. The optimizer is sensitive to the nature of the schedule it gets as input. Since LDS produces a series of reasonably good (but different) seed schedules, we can optimize each one, and in the end produce a schedule that is significantly shorter than we get by just optimizing the schedule given by following the heuristics exactly.

We can take this one step further and design the heuristic to avoid the kind of mistakes that the optimizer cannot fix.[3] This goes beyond the scope of the current paper, but it turns out that we can identify certain kinds of task-ordering mistakes that a simple right-left shift is unable to untangle. We can then generate heuristics that will generally avoid these mistakes. This may cause us to find worse unoptimized schedules, but better schedules after optimization.

7.7 Discussion

We have outlined an approach to RCPS problems that is based on using LDS to generate a series of "seed" schedules that are passed to an optimizer that can be seen as doing a kind of scheduling-specific local search. The results are currently the best known on problems of realistic size and character. Work continues on transitioning this technology to various application areas, and increasing the complexity of the constraints we can represent and effective optimize under.

[3]This idea came out of discussions with Matt Ginsberg.

Chapter 8

Procedural Attachment

For larger scheduling problems, like those discussed in chapter 7, it is impractical to encode as a constraint-satisfaction problem (CSP) or a satisfiability (SAT) problem. However, universal encodings, like CSP or SAT, have some clear advantages. If we have a problem encoded in such a form then we can immediately test out any one of a handful of modern, efficient algorithms with little or no programming time. Similarly, new algorithms can be implemented just once (and then be highly optimized), instead of being implemented once for each kind of specialized data structure required for a particular problem class.

Fortunately one can often hide the details of a domain (like scheduling) in a "black box" procedure that takes a partial assignment (*e.g.*, a partial schedule) and augments it with additional, forced, assignments. If we can do this successfully then we can implement each base search strategy (LDS, ISAMP, dynamic backtracking, TABLEAU, etc.) once and simply plug in a different black box propagator for each specific domain. This chapter addresses the key theoretical questions underlying such a modular architecture – specifically, what properties must the base level search algorithm and the propagator have to ensure that when plugged together they will be guaranteed to produce correct results. This work will appear in "Procedural reasoning in constraint satisfaction" by Jonsson and Ginsberg, at KR-96.

8.1 Introduction

For many constraint satisfaction problems there are simple functional relations (e.g. arithmetic equations) and simple subproblems (e.g. linear equations with unknowns) that can be done quickly, using simple algorithms. Needless to say, taking advantage of such methods can significantly decrease the time needed to find a solution. But doing so, without rewriting solvers and redoing correctness proofs, has proved to be difficult.

For general search engines, such as tableau [10], adding any kind of procedural reasoning can only be done by axiomatizing the functions and algorithms, and add the axioms as constraints to the problem. This requires introducing new variables and constraints, which increases the size of the problem. Furthermore, solving the problem is made harder since we end up using search to do simple things like arithmetic.

To use the procedures themselves, two approaches have been used in the past. One has been to rewrite the search engine, adding the procedure as part of the search mechanism. The other is to use procedural attachments [32] (see [54] for a description of procedural attachments), functions that a solver can call to directly calculate the value of certain variables.

Both solutions have their problems. The drawbacks of rewriting the solver each time we have a new technique to add, are many and obvious.

- It becomes almost impossible to do rapid prototyping, e.g. to compare different search engines, without sacrificing the efficiency of applicable reasoning algorithms.
- Specialized implementations tend to be brittle, they cannot handle small changes in the problem or additional constraint types.
- Without theoretical analysis, a search engine incorporating a number of additional algorithms, may not have the properties that were expected. For example, some search engines that use the pure literal rule, become incomplete when combined with a symmetry breaking algorithm.

The only advantage of specialized implementations is that clever coding can minimize the overhead of using reasoning procedures, such as unit propagation, by using data structures that are as efficient as possible for both solver and procedure.

The procedural attachment method has its problems too.

- Procedural attachments directly calculate values for problem variables, making it difficult to reuse a procedure in another search engine that has different data structures.
- The procedural attachments are basically functions, specifying the value of one variable, based on assignments to other variables. This makes it practically impossible to implement global algorithms like unit propagation.
- Procedural attachments do not take into account the effects they have on the search engine. Many authors assume the search engine is cooperative, while others assume the search engine does simple backtracking [48]. For instance, procedural attachments do not work with some dependency directed methods.

Instead of a handful of specially implemented solvers and a few safe procedural attachments, we need a general framework that allows us to combine any procedure with any search engine. The framework should precisely define what search engines and procedures are, and how they interface. This would allow procedures and search engines to be developed and implemented separately, at the same time making it trivial to use any applicable procedures with the search engine of choice. Finally, to guarantee that search engines and procedures work correctly together, we want to establish conditions that are sufficient to guarantee correctness, completeness and systematicity. These conditions should be as weak as possible.

In this paper we present such a general framework. In section 2 we present a formal description of search engines and extension procedures, and describe how the two are combined. In section 3, we prove easily satisfied conditions to be sufficient to guarantee systematicity and completeness. Concluding remarks are in section 4. In this extended abstract, all proofs, and a comprehensive example for the general reader, have been omitted.

8.2 Framework

8.2.1 Constraint Satisfaction Problems

A constraint satisfaction problem is a finite set of variables, a finite set of domain values for each variable, and a collection of constraints on the values assigned to the variables. This is formalized as follows:

Definition 8.2.1 *A* constraint satisfaction problem (CSP) *is a triple* (X, V, K)*, where* $X = \{x_1, \ldots, x_n\}$ *is a finite set of variables,* $V = \{V_{x_i}\}$ *is a set of domains for each variable, and* K *is a set of constraints. Each constraint is of the form* (Y, R)*, where* $Y = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ *and* $R \subseteq \prod_{\nu=1}^{k} V_{x_{i_\nu}}$.

To solve a constraint satisfaction problem, we need to find an assignment for each variable, such that all the constraints are satisfied. Formally:

Definition 8.2.2 *A* solution *to a* CSP (X, V, K) *is an n-tuple* $(v_{x_1}, \ldots, v_{x_n})$*, where* $n = |X|$*, such that:*

1. $v_{x_k} \in V_{x_k}$ *for* $k = 1, \ldots, n$*, and*

2. For any $(Y, R) \in K$ *with* $Y = \{x_{i_1}, \ldots, x_{i_k}\}$*, we have* $(v_{i_1}, \ldots, v_{i_k}) \in R$.

Few methods for solving constraint satisfaction problems are able to jump directly to a solution, most work with intermediate assignments that either do not assign values to all the variables, or do not satisfy all of the applicable constraints. These intermediate assignments are referred to as partial assignments:

Definition 8.2.3 *Given a* CSP (X, V, K) *with* n *variables, a* partial assignment *is a pair of m-tuples* $(x_{i_1}, \ldots, x_{i_m})$ *and* $(v_{i_1}, \ldots, v_{i_m})$*, where* $m \leq n$ *and* $v_{i_k} \in V_{x_{i_k}}$ *for* $k = 1, \ldots, m$*. A partial assignment will be called* consistent *if for any* $(Y, R) \in K$ *with*

$Y = \{x_{j_1}, \dots, x_{j_k}\} \subseteq \{x_{i_1}, \dots, x_{i_m}\}$*, we have* $(v_{j_1}, \dots, v_{j_k}) \in R$*. A partial assignment that is not consistent will be called* inconsistent.

If $p = \langle Y, W \rangle$ *is a partial assignment such that* $x_i \notin Y$*, we will call the partial assignment* $\langle Y \cup \{x_i\}, W \cup \{v\} \rangle$ *the result of* extending p by assigning the value v to the variable x_i. *We will denote this by* $p + \langle x_i, v \rangle$.

8.2.2 Search Engines

Constraint satisfaction problems are invariably solved with a search engine, an algorithm that searches for a solution by modifying a partial assignment. The prototypical example of a search engine is the depth-first search method:

Algorithm 8.2.4 To SOLVE a CSP (X, V, K), given a partial assignment p:

```
1   **if** $p$ is inconsistent, **return** failure
2   **if** $p$ is a solution, **return** $p$
3   **let** $x$ be a variable not assigned a value by $p$, **let** $E_x = V_x$
4   **if** $E_x$ is empty, **return** failure, **else** select $v \in E_x$
5   **if** SOLVE$(p + \langle x, v \rangle)$ succeeds, **return** result,
    **else** remove $v$ from $E_x$ and **goto** 4
```

Taking p to initially be the empty assignment, it is well known that the algorithm is systematic and complete.

The literature contains many such specific algorithms, but there is little discussion about search engines in abstract terms. Most researchers have focused on specific algorithms or presented ideas that intuitively apply to a certain class of algorithms. This, unfortunately, is not enough for our purpose, we need an abstract definition for search engines.

Let us examine how the depth-first algorithm above works. We notice that the E_x sets are used to control the search, by keeping track of which value assignments have been tried. In fact, the algorithm makes steady progress by gradually shrinking the set of partial assignments which are consistent with the E_x sets. The recursive nature of the algorithm controls the backtracking, by automatically returning to the calling function when a SOLVE function call fails. To sum up these observations, the E_x sets define the state, and the recursive calls determine how the states are updated.

Given this, the obvious way to define a search engine is to use a general state and an update function. This turns out to be too strong, since any Turing machine can be implemented within those parameters. Therefore we should focus on what differentiates search engines from other algorithms.

Consider any search engine that works by examining partial assignments, changing or extending them, trying to get to a solution. Such an engine will invariably have a set of partial assignments that may be considered in the near future, and will use this set to determine which partial assignment it will examine next. This is true for engines ranging from simple depth-first search and isamp [47] to dependency directed backtracking [69] and WSAT [62]. Based on these observations, we define a search engine as having a state that consists of a set of partial assignments, and a step function to update its state.

To formally define a search engine, we will need to identify certain concepts in constraint satisfaction. Given a CSP C, we will use $\mathcal{P}_C$ to denote the set of all possible partial assignments, consistent and inconsistent. The set of possible states will be $\mathcal{S}_C := 2^{\mathcal{P}_C}$, namely the set of all possible sets of partial assignments. Finally, to identify partial assignments that are solutions, we will use Γ_C to denote the set of all solutions to C. These concepts can be defined as follows:

Definition 8.2.5 *Given a* CSP *C, let $\mathcal{P}_C$ be the set of all partial assignments. Let $\mathcal{S}_C = 2^{\mathcal{P}_C}$. Let Γ_C be the set of all solutions to C.*

A search engine will have a state that is a member of $\mathcal{S}_C$, i.e. a set of partial assignments. The states will be updated using a successor function `succ` that maps one state to another. Associated with each state S will be a specific partial assignment, which corresponds to the partial assignment p in the depth first search engine above. This "current assignment" will be denoted by $\texttt{curr}(S)$. Both `succ` and `curr` will be partial functions, since a given search engine will only use a small subset of $\mathcal{S}_C$ to represent all of its states. Finally, a search engine will be problem-independent, so each of the functions `succ` and `curr` take the problem C as an argument. But since a search engine is only solving one problem at a time, we will omit the CSP parameter, for the sake of clarity.

Formally:

Definition 8.2.6 *A* search engine *is defined by a pair of partial functions* $(\texttt{succ}, \texttt{curr})$, *such that* $\texttt{succ} : \mathcal{S}_C \to \mathcal{S}_C$, *and* $\texttt{curr} : \mathcal{S}_C \to \mathcal{P}_C$. *Furthermore the partial functions satisfy:*

1. $\texttt{succ}(\mathcal{P}_C)$ *is defined.*
2. $\texttt{succ}(\emptyset) = \emptyset$
3. *If* $\texttt{succ}(S)$ *is defined,* $\texttt{curr}(S)$ *is defined.*
4. *If* $\texttt{succ}(S)$ *is defined,* $\texttt{succ}(\texttt{succ}(S))$ *is defined.*

This may look complicated, but the basic idea is simple. At each point, a search engine will have a state, consisting of a set of partial assignments that it views as candidates for examination. The function `succ` does one step in the search, updating the state by refining or changing the set of candidates. The function `curr` returns the partial assignment associated with that state, the "current" partial assignment. Given this, it is easy to see how $(\texttt{succ}, \texttt{curr})$ would typically be used to solve a CSP C:

Algorithm 8.2.7 To solve a CSP C, given a search engine $(\mathtt{succ}, \mathtt{curr})$:

```
1    let $S = \mathcal{P}_C$
2    if $S = \emptyset$, return failure
3    if $\mathtt{curr}(S) \in \Gamma_C$, return $\mathtt{curr}(S)$
4    let $S = \mathtt{succ}(S)$, goto 2
```

Conditions 1 and 2 in the definition merely state that the search engine has a zero-information initial state $\mathcal{P}_C$, and a final state $\emptyset$ that is used to terminate the search engine. Condition 3 means there is a "current" assignment associated with each state, and finally condition 4 says that once running, the search engine can keep running.

To further illustrate how this formalization works, let us look at depth-first search again. The function $\mathtt{curr}(S)$ returns one of the members of S that are at maximal depth and don't have parents in S. The update function $\mathtt{succ}(S)$ returns $S - \{\mathtt{curr}(S)\}$. See figure 8.1 for a small example.

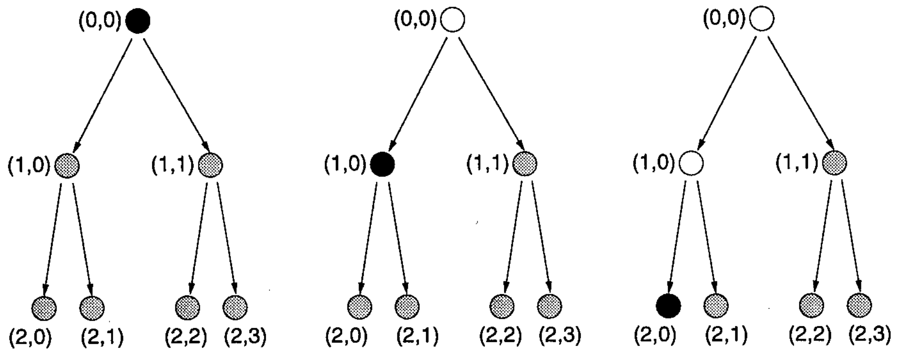

Figure 8.1: Depth first search, first three iterations. The non-white nodes are the partial assignments in S, the black node is $\mathtt{curr}(S)$.

The two properties of a search engine that we are most concerned about, are systematicity and completeness. These properties are easy to define in this setting:

Definition 8.2.8 *A search engine* $(\mathtt{succ}, \mathtt{curr})$ *is* complete *if, for any* CSP C*, such that* $\Gamma_C \neq \emptyset$*, there exists* $s \in \Gamma_C$ *and* $k \in \mathbb{N}$*, such that* $s = \mathtt{curr}(\mathtt{succ}^k(\mathcal{P}_C))$.

A search engine $(\mathtt{succ}, \mathtt{curr})$ *is* systematic *if, for any* CSP C*, and* $i, j \in \mathbb{N}$ *with* $i \neq j$*, either* $\mathtt{succ}^i(\mathcal{P}_C) \neq \mathtt{succ}^j(\mathcal{P}_C)$*, or both are empty.*

Completeness simply says that if there are any solutions to the problem, at least one of them will be found using the search engine. This is a weaker definition of completeness than many authors use (cf. [56] and [72]), but it is easy to see how an algorithm can be changed to find all solutions. Simply turn each solution found into a negative constraint, eliminating it from Γ_C, and start the search again.

The condition of systematicity states that the same state can never show up twice when a search engine is running. Since the number of possible states is finite, this also guarantees termination. This definition of systematicity differs from the standard definition, that the same path in the search tree is never examined twice. Our version allows the same paths to be explored repeatedly, but each time in a different global state. This new definition is better applicable to modern search engines, which may in fact search the same path more than once, e.g. iterative deepening [43] and dynamic backtracking [26].

8.2.3 Procedures

The mechanism we propose for procedural reasoning is simple, but powerful. Given a partial assignment to a CSP, the mechanism will either indicate which variable assignments are to be added or that the current partial assignment is a dead end. Almost any function and algorithm applicable to a CSP fits within this framework.

Let us formally define an extension procedure:

Definition 8.2.9 *Let C be a* CSP. *An* extension procedure *for C is a function $e : \mathcal{P}_C \to \mathcal{P}_C \cup \{\bot\}$, such that if $e(p) \neq \bot$, then $p \subseteq e(p)$.*

An extension procedure is correct *if for any $p \in \mathcal{P}_C$ and $s \in \Gamma_C$, such that p extends to s, $e(p) \subseteq s$.*

The correctness criteria states that if p can be extended to a specific solution, $e(p)$ can be extended to the same solution. This means that ways to extend p, other than through $e(p)$, do not lead to a solution.

8.2.4 Combining Engines and Procedures

Having formalized the concepts of search engines and extension procedures, let us turn to the task of combining the two. The main problem is that procedures can potentially return information that does not fit within the data structures used by the search engine, causing the search engine to fail. This is solved by allowing the search engine to choose a subset of the information returned by the extension procedure.

More formally, let us define $\mathcal{L}_C = \{S \in \mathcal{S}_C \mid \texttt{succ}(S) \text{ is defined}\}$, the set of all legal states for the search engine $(\texttt{succ}, \texttt{curr})$ working on problem C. As long as the search engine does not encounter a state $S \notin \mathcal{L}_C$, it will not fail while solving C (although it may return failure). So if e is an extension procedure, all we need to do is to require the result of invoking e to be within this set of legal states.

We can now formally define how a search engine uses an extension procedure:

Definition 8.2.10 *Given a search engine $(\texttt{succ}, \texttt{curr})$, a* CSP *$C$ and an extension procedure e for C, we define a* search engine using procedure e *as any function $\texttt{succ}_e : \mathcal{S}_C \to \mathcal{S}_C$ such that for any $S \in \mathcal{L}_C$, we have:*

1. $\mathtt{succ}(S) - \mathtt{elim}(e, \mathtt{curr}(S)) \subseteq \mathtt{succ}_e(S) \subseteq \mathtt{succ}(S)$
2. $\mathtt{succ}_e(S) \in \mathcal{L}_C$

where $\mathtt{elim}(e,p)$ is the set of nodes that can safely be eliminated from the search space, when e is invoked on the partial assignment p. The definition is valid, since there is always at least one legal state that satisfies it, namely $\mathtt{succ}(S)$. The only question remaining is what $\mathtt{elim}(e,p)$ is. Figure 8.2 will make it clear.

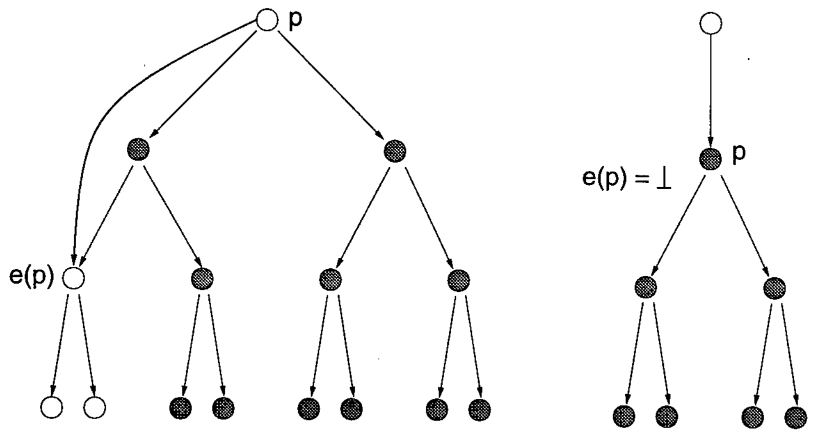

Figure 8.2: The set of nodes (colored gray) that can be eliminated from the search space when extension procedure e is invoked on partial assignment p.

If the procedure e is correct, extending the current partial assignment p to $e(p)$ means that the gray nodes in the left tree cannot be extended to a solution, making it safe to remove them from the search space. If we use $\mathtt{ext}(p)$ to denote the set of all possible ways to extend p, including p itself, the nodes that can be eliminated are those that appear in $\mathtt{ext}(p)$ but not in $\mathtt{ext}(e(p))$.

If the extension procedure finds that p is a dead end, i.e. it cannot be extended to a solution, it and every partial assignment that extends it, can safely be pruned from the tree. This set is exactly $\mathtt{ext}(p)$.

Therefore we define `elim` as follows:

Definition 8.2.11 *Given a partial assignment p, a correct extension procedure e,* $\mathtt{elim}(e,p)$*, the set of partial assignments that can be eliminated from the search space is:*

1. $\mathtt{elim}(e,p) = \mathtt{ext}(p) - \mathtt{ext}(e(p))$ *if* $e(p) \neq \perp$
2. $\mathtt{elim}(e,p) = \mathtt{ext}(p)$ *if* $e(p) = \perp$

By choosing a set that contains all, some or none of the partial assignments eliminated by the extension procedure, the search engine can fit the information from the extension procedures to its own data structure. In most cases a search engine will choose a minimal set for $\mathtt{succ}_e(S)$.

8.3 Theory

Having defined search engines, procedures and their interface, we now turn our attention to what properties a search engine will have, when used with a procedure. The reasons for doing this are clear. The framework allows search engines and procedures that are implemented separately, to be used together. The possibility of conflict is real, experiments have shown that sophisticated search engines may lose systematicity if care is not taken. Theoretical conditions that will prevent this possibility of conflict, are essential.

Given that procedures simply put us in a different part of the search space, the most obvious solution is to require the search engine to be complete and systematic, regardless of its starting point.

Definition 8.3.1 *A search engine* (succ, curr) *is* globally complete *if for any* CSP C*, and any* $S \in \mathcal{L}_C$*, such that* $S \cap \Gamma_C \neq \emptyset$*, there exists* $s \in S \cap \Gamma_C$ *and* $n \in \mathbb{N}$ *such that* $\texttt{curr}(\texttt{succ}^n(S)) = s$.

A search engine (succ, curr) *is* globally systematic *if for any* CSP C*, any* $S \in \mathcal{L}_C$*, and* $i, j \in \mathbb{N}$ *with* $i \neq j$*, either* $\texttt{succ}^i(S) \neq \texttt{succ}^j(S)$ *or both are empty.*

These are useful concepts that we will use later, but they are not sufficient to guarantee completeness and systematicity when an extension procedure is used. To see why, let us assume a search engine gradually expands its scope of candidates, e.g. some sort of iterative best first search. If an extension procedure happened to prune a state such that the result was a state encountered in an earlier iteration, the search engine would start to loop, losing both systematicity and completeness.

So we need to be a little bit more careful. The problem in this case turns out be that reducing the set of possible candidates can actually set us back. This can be avoided by requiring the search engine to satisfy $\texttt{succ}^k(S) \not\supseteq S$, for all $S \in \mathcal{L}_C$ and $k \in \mathbb{N}$. This is quite reasonable, most systematic search engines make progress by gradually (but not necessarily monotonically) reducing the set of partial assignments that may be looked at. Unfortunately this is too weak, figure 8.3 shows the problem that can be encountered.

The key observation that allows us to arrive at the correct condition, is that search engines are systematic for various different reasons. To take advantage of this observation, let us first note that global systematicity is equivalent to having an enumeration of the states so that each update moves to a lower numbered state. More formally:

Lemma 8.3.2 *A search engine* (succ, curr) *is globally systematic if and only if, for any* CSP C*, there exists a function* $\texttt{enum} : \mathcal{L}_C \to \mathbb{N}$*, such that* $\texttt{enum}(\texttt{succ}(S)) < \texttt{enum}(S)$*, for all* $S \in \mathcal{L}_C$.

Given this lemma, we can now formalize the condition we need to guarantee systematicity in search engines using procedures: Using a procedure must count as

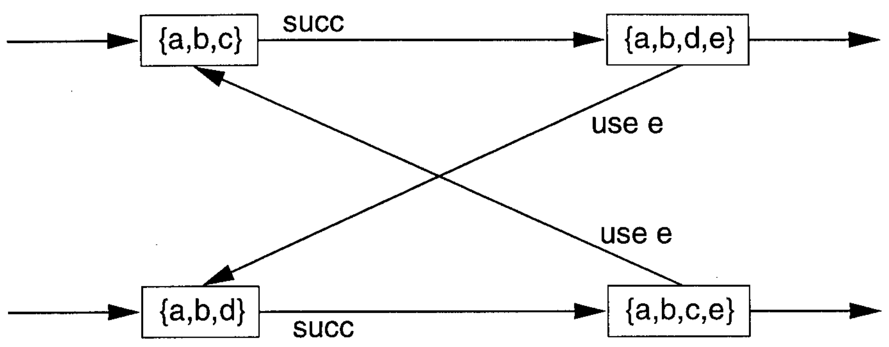

Figure 8.3: How a procedure may cause a loop in the search engine, even when $\texttt{succ}(S)$ is not a superset of S.

progress, i.e. if S' is the result of using procedure e in state S, we have $\texttt{enum}(S) > \texttt{enum}(S')$. Since procedures work by reducing the given state, we simply require that making a state smaller counts as progress, i.e. if $S' \subset S$ then $\texttt{enum}(S') < \texttt{enum}(S)$.

And when we have systematicity, it is easy to guarantee completeness: If a state contains solutions, either the current partial assignment must be a solution, or at least one solution remains in the updated state. This is enough, since a correct procedure will not eliminate any solutions.

To state our main theorem formally:

Theorem 8.3.3 *Let* $(\texttt{succ}, \texttt{curr})$ *be a search engine such that for any* CSP C *it satisfies:*

1. *There exists a function* $\texttt{enum} : \mathcal{L}_C \to \mathbb{N}$*, such that for any* $S, S' \in \mathcal{L}_C$*, with* $S' \subseteq \texttt{succ}(S)$ *we have* $\texttt{enum}(S') < \texttt{enum}(S)$*, or* S' *and* S *are both empty.*
2. *For any* $S \in \mathcal{L}_C$*, such that* $S \cap \Gamma_C \neq \emptyset$*, either* $\texttt{curr}(S) \in \Gamma_C$ *or* $\texttt{succ}(S) \cap \Gamma_C \neq \emptyset$.

Then, for any correct extension procedure e, the search engine using procedure e, i.e. $(\texttt{succ}_e, \texttt{curr})$*, satisfies the same conditions.*

The conditions in this theorem guarantee global systematicity and global completeness:

Lemma 8.3.4 *If a search engine* $(\texttt{succ}, \texttt{curr})$ *satisfies condition 1 in Theorem 8.3.3, it is globally systematic. If a search engine satisfies conditions 1 and 2 in Theorem 8.3.3, it is globally complete.*

It is worth noting that the converse does not hold, the conditions stated in the theorem are strictly stronger than global systematicity and global completeness.

Using this last lemma and the main theorem, we get exactly what we wanted, namely conditions that guarantee completeness and systematicity for a search engine using a correct procedure.

Corollary 8.3.5 *If* $(\texttt{succ}, \texttt{curr})$ *satisfies the conditions in Theorem 8.3.3, and* e *is a correct procedure, then* $(\texttt{succ}_e, \texttt{curr})$ *is complete and systematic.*

The conditions of Theorem 8.3.3 are fairly weak, in the sense that almost any known systematic search engine can be made to satisfy those conditions. In fact, most systematic search engines are covered by the following corollary:

Corollary 8.3.6 *Let* $(\texttt{succ}, \texttt{curr})$ *be a search engine that is monotonic in the sense that for any* CSP C *we have* $\texttt{succ}(S) \subset S$ *for all* $S \in \mathcal{L}_C$*. If* $(\texttt{succ}, \texttt{curr})$ *satisfies condition 2 in Theorem 8.3.3, then* $(\texttt{succ}_e, \texttt{curr})$ *is systematic and complete, for any correct extension procedure* e*.*

Finally, let us note that Theorem 8.3.3 allows us to use as many correct extension procedures as desired, since the conditions remain unaffected by the addition of extension procedures.

8.4 Discussion

We have presented a theoretically sound framework that allows search engines to use extension procedures to speed up the search. We have also presented conditions are sufficient to guarantee correctness, completeness and systematicity, regardless of the number of correct procedures the search engine uses.

The theoretical results are based on having found the correct definition for each entity in the framework. As we have already pointed out, certain concepts (e.g. abstract search engine) needed to be invented, while others (e.g. systematicity and completeness) were modified to be more generally applicable. This theoretical framework provides a foundation, on which further theoretical analysis of search engines can be built, even if the framework appears obvious in retrospect.

Finally, the framework allows us to implement CSP solvers that can access a library of extension procedures that speed up the search. As long as the solvers satisfy the conditions expressed in Theorem 8.3.3, and each of the procedures is correct, the system will remain complete and systematic. This is invaluable in the search for better search engines and more powerful reasoning methods, in addition to moving us closer to an efficient, general problem solver.

Chapter 9

Broader Issues in Planning

Although the main thrust of our work under this award has been on scheduling and combinatorial optimization, we have continued to concern ourselves with the broader issues involved in building practical planning systems. Having very efficient and effective search-control techniques clearly helps in the development of planning systems, but practical planning requires more than this. Realistic planning problems don't fall to idealized solution methods. Assuming you can exhaustively consider all contingencies and totally specify exactly what will happen at every instant doesn't work. Practical planners must attend to the most important things first, and be prepared to give their current best guess at a solution when a solution is required: a fully-thought-out solution derived too late is no solution at all. Similarly, the traditional approach in generative planning, which presumes that the plan will be executed exactly as described, with nothing else being done (even by other agents), clearly doesn't jibe with the experience of planning in the real world.

We have discovered that an appropriate view of modality points the way toward solutions to both of these problems, by providing a well-grounded mechanism for interrupting and resuming computation in favor of higher-level goals and by making possible a more natural and flexible representation of plans. In this chapter we briefly summarize our work on these more fundamental issues in automated reasoning. We first describe various concepts of modality and a unifying semantic framework that we have discovered that underlies them. Then, we discuss how modalities can serve as interruption markers. These markers allow the reasoner to substitute an approximation for the result of evaluating the modality if time pressure dictates, and later return to refine its answers by actually continuing the evaluation. Finally, we describe a general framework for planning – "approximate planning" – that we have developed using these ideas. This work is described in detail in two papers by Ginsberg: "Modality and interrupts" in the *Journal of Automated Reasoning*, and "Approximate planning" in *Artificial Intelligence.*

Modal operators are used in a variety of ways in AI, including reasoning about knowledge and belief, reasoning about time, and applications to nonmonotonic inference [33, 65, 53, and others]. The semantics assigned to a particular modal operator are usually determined using a scheme due to Kripke [44] that is based on the notion of possible worlds linked by an accessibility relation. Moore, however, needs to define his own semantics in [53] in order to establish the desired link between a modal operator of knowledge and existing ideas in nonmonotonic reasoning.

Our work on modality has shown that Moore's and Kripke's ideas can be unified into a single approach if we view modal operators not in terms of possible worlds, but as mappings on the truth values assigned to various sentences. Thus the modal operator L, where Lp means, "I know that p," simply assigns the truth value true to Lp if p is known to be true, and assigns Lp the value false if p is either known to be false or is neither known to be true nor to be false.

This approach is made possible by the fact that we have worked with a formal system that explicitly allows us to label sentences with values other than the conventional *true* and *false*. The description in the previous paragraph, for example, implicitly took advantage of the availability of a label indicating that p was "unknown" – not known to be true nor to be false. Somewhat more precisely, the truth values are taken not from the two-point set $\{t, f\}$ but from a larger set known as a *bilattice*. Formally, then, a modal operator can be viewed as a function on the elements of the bilattice of truth values.

We have shown that first-order logic, Kripke's work, and Moore's construction are all special cases of our general approach. Moreover, our ideas also let us extend existing notions, not just combine them under a single formal framework. As an immediate example, it becomes possible to define modal operators that combine the features of Kripke's and of Moore's. We have also implemented an automated theorem prover that handles modal operators of the sort we have defined. The implementation extends existing work on *stratification,* [1, 23, 74, 75] which can be used to compute the consequences of some autoepistemic theories, to our more general setting.

This truth-value-mapping approach to modality allows "modality" to be interpreted very generally. For example, causality, temporal notions, and even negation can be viewed as modal operators. This, in turn, has suggested a new role for modalities in the control of computation in resource-limited reasoning. We argue that modal operators can improve the usefulness of declarative systems by marking points at which inference can be suspended and an approximate answer returned. Viewing the modal operators truth-functionally, their correct evaluation requires a recursive call to the theorem prover to evaluate the sentential arguments of any particular modal expression. If the computational resources needed by this recursive call are not available, we can instead use the values found by simply searching the database for these sentences.

Thus, for example, if $\mathbf{L}_{System}\mathbf{L}_{Jones}\,codes$ is interpreted as "The system knows that 'Jones knows the launch codes.'," the reasoner may initially decide to approximate the results of trying to prove that it knows that Jones knows the codes by whether it *explicitly* knows. Later, should sufficient resources be available, the reasoner could go back and devote more effort to proving it, perhaps again approximating the inner modality ($\mathbf{L}_{Jones} codes$) with a check for whether *codes* is explicitly in the system's representation of Jones' knowledge base, KB_{Jones}. Finally, an attempt could be made to prove *codes* from KB_{Jones}.

We have characterized when it makes sense to interrupt computation at a modality, and developed an anytime declarative procedure that responds to queries by repeatedly calling a conventional first-order theorem prover and then accumulating the results. Approximate answers are computed quickly and gradually refined if time permits. Although the final answer returned by the system may appear to be changing in a nonsystematic way, the quality of the answer improves uniformly as additional information is considered.

To apply these notions to planning, however, we must be able to handle temporal notions. These can also be captured in our setting. One conventional approach to temporal reasoning involves treating sentences in the representation language as themselves objects in that language. For example, in order to say that some sentence p holds at a time t, we might actually write

$$\texttt{holds}(p, t). \tag{9.1}$$

It seems more natural to treat `holds` as a modal operator, although this requires us to deal with the temporal variable appearing in (9.1). We can do this by temporally extending the bilattice B with which we are working, replacing it with B^T, where T is the set of time points in our temporal language. Thus we label a sentence not with an element of B, but with a *function* that gives its truth value as a function of time. Other modalities in this setting include `delay` (which separates the occurrence of an action from its effects) and `propagate` (which is responsible for describing the frame axiom in our setting). All of these modalities underlie the implementation of the approximate planning framework, which we now describe.

Just as it is typically computationally infeasible to completely ground out every inference in realistic large reasoning problems, it is infeasible to plan every single thing that will happen during an attempt to achieve a goal. The traditional notion of a "plan" in the generative planning community is remarkably different from what people generally mean by "plans". When we talk about a plan for achieving a goal, we typically mean not one course of action but many. As an example, if our plan for suppressing a rebel uprising is to establish a beachhead and then advance inland, we hardly mean that these are the only actions to be taken. We may also plan on conducting a variety of unrelated military actions elsewhere, on briefing the chain of command as the plan proceeds, and so on.

In fact, the plan "establish a beachhead and advance inland" might be represented something like this:

$$[\ldots\texttt{beachhead}\ldots\texttt{advance}\ldots] \tag{9.2}$$

where the ellipses denote currently undetermined action sequences that might be interspersed into the above plan. If we need to advance *immediately* after the beachhead is established, we might write

`[...beachhead advance...]` (9.3)

dropping the second set of ellipses.

There are, of course, many instance of (9.2) that are unsatisfactory. Perhaps we abandon the beachhead after establishing it, or dump all of our fuel into the ocean. The nub of building a planning system that preserves our intuitive view of what a plan is is finding an answer to the question, "In what sense can we say that (9.2) is our plan, when so many things can go wrong?"

The conventional approach to this problem is to deal not with plans such as that appearing in (9.2), but with far more specific plans such as

`[beachhead status-report move-destroyer advance status-report]` (9.4)

where there are guaranteed to be no extraneous actions that might interfere with our achieving our goal. But from a practical point of view, the plan (9.4) is nearly worthless, since it is almost inconceivable that we execute it exactly as written, but in every other case there are no guarantees that the plan is reasonable: the justification for the plan presumes the whole plan's being executed exactly as written.

There are many other examples of the inadequacy of fully specified plans. If we intend to construct plans by retrieving them from a library of known solutions to similar problems (so-called *case-based* planning [34]), it is important that the plans in the library include some measure of flexibility. After all, it is unlikely that the new situation in which we find ourselves will be an exact match for the situation in which the plan was constructed.

Approximate planning is an attempt to formalize the ideas that are implicit in plans like (9.2). Plans dictate what must be done, but generally do not preclude other things being done. They provide recipes that will generally achieve the goal, but there are generally pathological execution sequences (involving extra actions or events) that may fail. They may explicitly rule out certain "extra" actions that would otherwise break the plan, but eventually they ground out in some level of "approximate correctness" that suffices for the problem at hand.

Our successful formalization of approximate planning involved developing a mechanism for expressing plans that can have new actions added to them in arbitrary ways but that can still express the immediacy requirements of a plan such as (9.3), and then defining conditions under which a plan "approximately" achieves a goal. The basic idea here is that a plan P is approximately correct if most instances of P that could actually be executed do indeed achieve the goal. We formalized this by introducing the idea of an exception to a plan and formalizing conditions under which plans hold sufficiently frequently that we are prepared to treat them as approximately correct. The intuitions underlying this formalization are based on the analytic notion of one set being of measure zero in another.

An anytime planning process then consists of finding a plan that approximately achieves the objectives. If time allows, the planner may then look for instantiations of the plan that, because of pathological actions added in, amount to approximate plans to fail to achieve the objective. The original plan can then be refined to exclude these. These classes can be further refined as time allows, iterating the planning process to achieve arbitrarily fine-grained plan sets.

Our ability to plan for conjunctive goals rests on similar ideas. When possible, it is important that we plan for conjuncts separately and then merge the results; this appears to require that the solutions to the individual conjuncts be plan schemas like (9.2). Planning for conjuncts separately enables us to take computational advantage of the relative benevolence of our environment reflected by the frame assumption – we can typically achieve one subgoal and then not worry about it while we work on other things. Merging the plans for the different subgoals requires the flexibility implicit in (9.2) but not in (9.4).

Related problems appear in plan debugging. If a human planner discovers a bug in one portion of a plan to achieve a complex goal, the typical response is to identify a small portion of the analysis to which the impact of the bug is restricted, and to then plan around the problem. That we can make modifications that address the bug without destroying the effect of the original plan depends on our common sense ability to construct and manipulate plans like (9.2) – plans that, while not succeeding universally, generally suffice.

The framework also facilitates mixed-initiative planning, since an externally provided plan for part of an objective can be integrated in the same way the independent plans generated for conjunctive subgoals are. The merging process makes sure that the plans are sufficiently compatible that the resulting plan remains approximately correct.

We built a prototype planner based on these ideas. The implementation exploits the fact that it is theoretically possible to plan for conjuncts separately using this approach, allowing problems to be decomposed into smaller subproblems. The notions of modality described earlier underlie our implementation of approximate planning, allowing for declarative representations of the system's knowledge about the effects of actions. We can thus use the declarative mechanisms that exist in the bilattice framework to manipulate the plan sets in question. This means that all we need do is provide a declarative description of action. We do not need to construct a special-purpose planner but can instead employ a general multivalued theorem prover [24, 27].

The notion of anytime refinement of plans that underlies approximate planning turns out to be the same as that of anytime refinement of modalities: the underlying assumption that what happens between specified actions "approximately" won't break the plan is handled by a modality that propagates the world state forward. Full evaluation of the modality involves trying to break the plan, repair it, break the repair, etc. For anytime evaluation, it may suffice to simply assume that nothing goes wrong, and refine (patch) later if time allows.

We believe that the understanding of modality we have developed, with its concomitant notions of interruptibility, will eventually play a key role in building planners (and reasoning systems in general) that have acceptable behavior, both computationally and epistemologically.

Chapter 10

Conclusion

The original dynamic backtracking algorithm was motivated by the observation that search algorithms solving crossword puzzles appeared to repeatedly erase, and then rediscover, solutions to parts of the puzzle. Over the course of this project the dynamic backtracking algorithm has been developed theoretically, and has been tested against a variety of academic and industrial problems. Out of this have come three novel algorithms that are now sufficiently mature to be applied to real problems: limited discrepancy search (LDS), relevance-bounded learning, and doubleback optimization. Equally important, our understanding of search in general, and dynamic backtracking in particular, has matured. In this concluding chapter we survey the types of problems for which these new algorithms are applicable, discuss our broader view of search control, and indicate the paths we expect the research to take from here.

Limited-discrepancy search is arguably the most significant new algorithm to come out of this project. Its significance lies not in its sophistication or mathematical depth, but in its range of applicability and in the ease with which it can be added to existing systems. It is sometimes said that "NP-completeness" matters only to academics: in the "real world" no one ever tells the plant manager that scheduling is intractable and as a result he goes again and generates schedules that he is happy with. If we look closely at how he does this, we find that he applies various heuristics and rules-of-thumb that he has learned from years of experience. When automated systems are built, the most practical approach has often been to build these heuristics into the system. Among other advantages, this makes the automated system more likely to generate schedules that are "like" those that the manager is used to seeing.

It is tempting to take such an automated system and make it systematic, or, equivalently, modify it to generate provably optimal solutions. This is usually done by augmenting the heuristic scheduler with some sort of branch-and-bound or chronological backtracking mechanism. In some cases this works, but more often it is an uphill battle, for various good reasons. One of the main reasons is that, as discussed in chapter 6, for medium to large problems, adding chronological backtracking to a good heuristic method is very unlikely to yield a significant improvement in solution quality.

LDS, on the other hand, is a systematic technique in the limit, but with limited computational resources it quite successfully searches "near" an existing heuristic. It can thus be easily added on top of existing heuristic techniques, and can be expected to yield a significant improvement in solution quality. The potential domain of applicability of LDS is thus any combinatorially difficult problem currently solved using heuristics without search. TPFDDs are an obvious example, and we have been working with Kestrel on adding LDS to KTS. In addition there are many other applications in military and industrial domains.

The reason that LDS works is simple. Heuristics make mistakes; if they made no mistakes we would call them algorithms. In most domains, however, we can reasonably assume that heuristics make relatively few mistakes. LDS searches by first assuming that the heuristic has made no mistakes, then that the heuristic has made at most one mistake, then two mistakes, etc., gradually widening the search until, eventually, the entire space is covered. The theme of finding and correcting mistakes in heuristically generated solutions is something we see repeated in each of the three successful algorithms developed in this project.

Doubleback optimization (DBO) is superficially quite different from LDS. Its applicability is limited to resource-constrained project scheduling (RCPS) in which the objective is to minimize the time to completion. However, for large RCPS problems doubleback optimization seems to be able to produce solutions that are of considerably better quality than a human expert could generate. In fact, on large benchmark problems relevant to aircraft manufacture (discussed in chapter 7) LDS and doubleback optimization together produce the best schedules currently known. RCPS may appear to be a limited domain, but it is essentially the domain served by the widely used "resource leveler" in `Microsoft Project`$^{\text{TM}}$. We have recently connected the optimizer to `Microsoft Project`, and are generally able to produce much better schedules than Microsoft's leveler, moreover, the improvement seems to grow with problem size. Using this tie to `Microsoft Project`, the optimizer can be easily tested on potential applications and can be deployed with a COTS interface.

One distinguishing feature of DBO is that it makes quite large jumps through the search space. That is, if we look at schedules before and after DBO executes they are very different (especially compared to a more traditional local search-based optimizer). For example, a single pass of DBO might move thirty or forty tasks from early in the schedule to fill in holes in the night shifts late in the schedule, and compress the schedule to take advantage of the resources that this frees up. In a sense DBO is correcting systematic or distributed mistakes throughout the schedule in a single pass (in this case the distributed mistake is the failure to put off tasks that can be done in the later night shifts). This complements LDS' ability to fix more subtle "spot" mistakes in the heuristic.

The third mature technique is now called *relevance-bounded learning* [2] in the literature, but we often refer to it as *dynamic backtracking light.* One lesson we have learned over the course of this project is that dynamic backtracking differs from previous algorithms in two orthogonal directions. First, like k-bounded learning or dependency-directed backtracking, dynamic backtracking learns from mistakes by generating nogoods. Unlike these techniques, however, it has a notion of "relevance" and it throws out nogoods when they become irrelevant. Second, and separately, dynamic backtracking allows lateral moves through the search tree. That is, decisions do not have to be revised chronologically, but rather can be revisited in a more flexible order. This gives dynamic backtracking the ability to make local-search-like moves, while maintaining systematicity. Adding lateral moves was the primary focus of the original grant proposal, and we still expect lateral moves to be critical, but relevance-bounded learning has proven easier to merge into larger systems.

Perfect relevance learning would involve keeping only those nogoods that will apply later in the search, and throwing out any nogood after its last application. Obviously we cannot do this. Even if we could there is no guarantee of polynomial memory usage since there may be an exponential number of future nodes in the search tree. The relevance-bounded learning used in dynamic backtracking is more modest. Whenever an infeasible node is reached, we choose a variable to revalue, and record a nogood as a justification for its new value. For example, if the nogood generated at the infeasible node is $\overline{air\_attack} \wedge \overline{sea\_attack} \wedge \overline{ground\_attack}$ (*i.e.*, some sort of attack is necessary) and the variable chosen is *air_attack*, then the new justification is:

$$(\overline{sea\_attack} \wedge \overline{ground\_attack}) \rightarrow air\_attack$$

This justification is kept and if at some future point in the search $\overline{sea\_attack} \wedge \overline{ground\_attack}$ becomes true again, then *air_attack* is immediately set true. However, if a new justification for *air_attack* is eventually learned, then the old justification is erased, and the new justification takes its place. Since only one justification is kept for each variable, memory usage is guaranteed to be polynomial.

A version of TABLEAU using relevance bounded learning is currently the best technique we have for propositional encodings of planning problems. Furthermore, as discussed in the introduction, it also gives quite good performance on academic scheduling problems. In fact, the RBL version of TABLEAU is currently the first algorithm we would try on a new satisfiability problem of unknown structure.

RBL works because, like dependency-directed backtracking, it is able to learn from its mistakes. However, unlike dependency-directed backtracking, RBL is guaranteed to use a polynomial amount of memory, which is critical on larger problems.

Dynamic variable reordering, or equivalently moving laterally through the search space, is a more complex technique to apply. When it is combined with essential basic) techniques like value propagation, one tends to run into the problems described in chapter 4. These problems have been found to occur in both academic problems and in industrial scheduling problems [3]. Nevertheless, in the longer term we view allowing lateral moves while maintaining systematicity as critical to solving large, hard optimization problems. Clearly one needs to be able to move easily from one portion of the search space to another. When people talk about debugging or patching an existing plan, they are generally referring to the creation of another plan that is similar to – but not the same as – an existing one. Changing only the few relevant features typically corresponds to a large transverse move in the search tree being investigated.

This brings us back to our broader vision of search and combinatorial optimization. Search problems are hard because they involve a large number of interacting decisions and it is extremely difficult to make them all correctly. As a result, we can generally assume (as each of these three algorithms do) that we will be working from a candidate solution that contains sub-optimal decisions – *i.e.,* mistakes. Our "big picture" goal is to be able to correct mistakes, including distributed mistakes, and so arrive at optimal solutions. Our research on dynamic backtracking suggests that this requires the ability both to reconsider any decision at any time, and to learn from mistakes without being overwhelmed by bookkeeping.

The algorithms developed under this project take a number of steps toward this goal. LDS provides a way of solving the odd mistake made by a good but imperfect heuristic, DBO allows distributed mistakes to be corrected, and RBL lets the system learn from mistakes without drowning in bookkeeping. Full dynamic backtracking allows considerable freedom in the order in which decisions are reconsidered without giving up guarantees of systematicity and optimality. Abstracting and generalizing these attributes to develop a single systematic search control mechanism presents a number of technical problems that provide focus for our ongoing research.

Bibliography

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, 1987.

[2] R. Bayardo and D. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[3] M. Boddy, Aug. 1996. personal communication.

[4] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[5] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 163–169, 1991.

[6] C. Cheng and S. Smith. Generating feasible schedules under complex metric constraints. In *Proc. of the Twelvth National Conference on Artificial Intelligence*, 1994.

[7] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *JACM*, 35:759–768, 1988.

[8] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. 11th AAAI*, 1993.

[9] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. 12th AAAI*, 1994.

[10] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, 1993.

[11] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3sat. *Artificial Intelligence*, 81:13–59, 1996.

[12] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[13] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. In *CACM*, pages 394–397, 1962.

[14] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[15] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Assoc. Comput. Mach.*, 7:201–215, 1960.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[17] J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.

[18] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 271–277, 1989.

[19] B. Fox. An algorithm for scheduling improvement by scheduling shifting. Technical Report 96.5.1, McDonnell Douglas Aerospace - Houston, 1996. McDonnell Douglas has applied for a patent on this work.

[20] M. Garey and D. Johnson. *Computers and Intractability.* W.H. Freeman and Company, New York, 1979.

[21] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[22] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1979. Available as Technical Report CMU-CS-79-124.

[23] M. Gelfond. On stratified autoepistemic theories. Technical report, University of Texas at El Paso, 1986.

[24] M. L. Ginsberg. The MVL theorem proving system. *SIGART Bulletin*, 2(3):57–60, 1991.

[25] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[26] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[27] M. L. Ginsberg. User's guide to the MVL system. Technical report, University of Oregon, 1993.

[28] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210–215, 1990.

[29] M. L. Ginsberg and W. D. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.

[30] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 226–237, 1994.

[31] R. Goldman, M. Boddy, and M. Ringer. Constraint-based scheduling for batch manufacturing. In *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 63–70, 1996.

[32] C. C. Green. *The application of theorem proving to question answering systems.* PhD thesis, Stanford University, Stanford, CA, 1969.

[33] J. Y. Halpern and Y. Moses. A guide to the modal logics of knowledge and belief: Preliminary draft. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 480–490, 1985.

[34] K. J. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[35] W. Harvey. Search and job shop scheduling. Technical Report CIRL TR 94-1, CIRL, University of Oregon, 1994.

[36] W. Harvey. *Nonsystematic Backtracking Search.* PhD thesis, Stanford University, 1995.

[37] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 607–613, 1995.

[38] A. K. Jónsson and M. L. Ginsberg. Experimenting with new systematic and nonsystematic search procedures. In *Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems*, 1993.

[39] A. K. Jonsson and M. L. Ginsberg. Experimenting with new systematic and nonsystematic search techniques. In *Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems*, Stanford, California, 1993.

[40] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[41] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1982.

[42] K. Konolige. Easy to be hard: Difficult problems for greedy algorithms. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Bonn, Germany, 1994.

[43] R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[44] S. A. Kripke. Semantical considerations on modal logic. In L. Linsky, editor, *Reference and Modality*, pages 63–72. Oxford University Press, London, 1971.

[45] P. Langley. Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pages 145–152. Morgan Kaufmann, 1992.

[46] P. Langley. Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, 1992.

[47] P. Langley. Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pages 145–52. Morgan Kaufmann, 1992.

[48] J. Maluszynski, S. Bonnier, J. Boye, F. Kluzniak, A. Kagedal, and U. Nilsson. Logic programs with external procedures. In K. R. Apt, J. de Bakker, and J. J. M. M. Rutten, editors, *Logic Programming Constraints, Functions, and Objects*, pages 21–48. MIT Press, Cambridge, MA, 1993.

[49] D. A. McAllester. Partial order backtracking. `ftp.ai.mit.edu:/pub/dam/dynamic.ps`, 1993.

[50] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 17–24, 1990.

[51] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

[52] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, 1992.

[53] R. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[54] K. L. Myers. Hybrid reasoning using universal attachment. *Artificial Intelligence*, 67:329–375, 1994.

[55] C. B. P. Purdom and E. Robertson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–114, 1981.

[56] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[57] N. Sadeh. Look-ahead techniques for micro-opportunistic job shop scheduling. Technical Report CMU-CS-91-102, School of Computer Science, Carnegie Mellon Univ., 1992.

[58] R. Seidel. A new method for solving constraint satisfaction problems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338–342, 1981.

[59] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 290–295, 1993.

[60] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings 1993 DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*, 1993.

[61] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings 1993 DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*, 1993.

[62] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings 1993 DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*, 1993.

[63] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[64] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[65] Y. Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence.* MIT Press, Cambridge, MA, 1988.

[66] D. E. Smith and M. R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):171–215, 1985.

[67] S. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proc. of the Eleventh National Conference on Artificial Intelligence*, 1993.

[68] S. F. Smith and C.-C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 139–144, 1993.

[69] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–96, 1977.

[70] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[71] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[72] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, 1993.

[73] R. Vaessens, E. Aarts, and J. Lenstra. Job shop scheduling by local search. Technical Report COSOR 94-05, Eindhoven University of Technology, 1994.

[74] A. Van Gelder. The alternating fixpoint of logic programs with negation: Extended abstract. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1989.

[75] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *J. Logic Program.*, 6:109–133, 1989.

[76] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, San Mateo, CA, 1988.

[77] C. P. Williams and T. Hogg. Using deep structure to locate hard problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[78] Y. Xiong, N. Sadeh, and K. Sycara. Intelligent backtracking techniques for job shop scheduling. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, Boston, MA, 1992.

[79] R. Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 46–51, 1990.

[80] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, 1988.

[81] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, 1988.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-47119

DISTRIBUTION LIST

addresses	number of copies
NORTHRUP FOWLER III ROME LABORATORY/C3C 525 BROOKS RD ROME NY 13441-4505	10
UNIVERSITY OF OREGON ATTN: MATTHEW L. GINSBERG COMPUTATIONAL INTELL RESEARCH LAB 1269 UNIVERSITY OF OREGON EUGENE OR 97403-1269	5
ROME LABORATORY/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ROME LABORATORY/C3AB 525 BROOKS RD ROME NY 13441-4505	1
ATTN: RAYMOND TADROS GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1
AFIT ACADEMIC LIBRARY/LDEE 2950 P STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1

OL AL HSC/HRG, BLDG. 190 2698 G STREET WRIGHT-PATTERSON AFB OH 45433-7604	1
US ARMY STRATEGIC DEFENSE COMMAND CSSD-IM-PA P.O. BOX 1500 HUNTSVILLE AL 35807-3801	1
NAVAL AIR WARFARE CENTER 6000 E. 21ST STREET INDIANAPOLIS IN 46219-2189	1
COMMANDER, TECHNICAL LIBRARY 474700D/C0223 NAVAIRWARCENWPNDIV 1 ADMINISTRATION CIRCLE CHINA LAKE CA 93555-6001	1
SPACE & NAVAL WARFARE SYSTEMS COMMAND (PMW 178-1) 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	2
COMMANDER, SPACE & NAVAL WARFARE SYSTEMS COMMAND (CODE 32) 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
CDR, US ARMY MISSILE COMMAND RSIC, BLDG. 4484 AMSMI-RD-CS-R, DOCS REDSTONE ARSENAL AL 35898-5241	2
ADVISORY GROUP ON ELECTRON DEVICES SUITE 500 1745 JEFFERSON DAVIS HIGHWAY ARLINGTON VA 22202	1
REPORT COLLECTION, CIC-14 MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1

AEDC LIBRARY TECHNICAL REPORTS FILE 100 KINDEL DRIVE, SUITE C211 ARNOLD AFB TN 37389-3211	1
COMMANDER USAISC ASHC-IMD-L, BLDG 61801 FT HUACHUCA AZ 85613-5000	1
US DEPT OF TRANSPORTATION LIBRARY FB10A, M-457, RM 930 800 INDEPENDENCE AVE, SW WASH DC 22591	1
AIR WEATHER SERVICE TECHNICAL LIBRARY (FL 4414) 859 BUCHANAN STREET SCOTT AFB IL 62225-5118	1
AFIWC/MSO 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
SOFTWARE ENGINEERING INSTITUTE CARNEGIE MELLON UNIVERSITY 4500 FIFTH AVENUE PITTSBURGH PA 15213	1
NSA/CSS K1 FT MEADE MD 20755-6000	1
DCMAO/WICHITA/GKEP SUITE B-34 401 N MARKET STREET WICHITA KS 67202-2095	1

PHILLIPS LABORATORY PL/TL (LIBRARY) 5 WRIGHT STREET HANSCOM AFB MA 01731-3004	1
THE MITRE CORPORATION ATTN: E. LADURE D460 202 BURLINGTON RD BEDFORD MA 01732	1
OUSD(P)/DTSA/DUTD ATTN: PATRICK G. SULLIVAN, JR. 400 ARMY NAVY DRIVE SUITE 300 ARLINGTON VA 22202	2
DR JAMES ALLEN COMPUTER SCIENCE DEPT/BLDG RM 732 UNIV OF ROCHESTER WILSON BLVD ROCHESTER NY 14627	1
DR YIGAL ARENS USC-ISI 4676 ADMIRALTY WAY MARINA DEL RAY CA 90292	1
DR RAY BAREISS THE INST. FOR LEARNING SCIENCES NORTHWESTERN UNIV 1890 MAPLE AVE EVANSTON IL 60201	1
DR MARIE A. BIENKOWSKI SRI INTERNATIONAL 333 RAVENSWOOD AVE/EK 337 MENLO PRK CA 94025	1
DR PIERO P. BONISSONE GE CORPORATE RESEARCH & DEVELOPMENT BLDG K1-RM 5C-32A P. O. BOX 8 SCHENECTADY NY 12301	1
MR. DAVID BROWN MITRE EAGLE CENTER 3, SUITE 8 O'FALLON IL 62269	1

DR MARK BURSTEIN BBN SYSTEMS & TECHNOLOGIES 10 MOULTON STREET CAMBRIDGE MA 02138	1
DR GREGG COLLINS INST FOR LEARNING SCIENCES 1890 MAPLE AVE EVANSTON IL 60201	1
DR STEPHEN E. CROSS SCHOOL OF COMPUTER SCIENCE CARNEGIE MELLON UNIVERSITY PITTSBURGH PA 15213	1
DR THOMAS CHEATHAM HARVARD UNIVERSITY DIV OF APPLIED SCIENCE AIKEN, RM 104 CAMBRIDGE MA 02138	1
MS. LAURA DAVIS CODE 5510 NAVY CTR FOR APPLIED RES IN AI NAVAL RESEARCH LABORATORY WASH DC 20375-5337	1
DR THOMAS L. DEAN BROWN UNIVERSITY DEPT OF COMPUTER SCIENCE P.O. BOX 1910 PROVIDENCE RI 02912	1
DR WESLEY CHU COMPUTER SCIENCE DEPT UNIV OF CALIFORNIA LOS ANGELES CA 90024	1
DR PAUL R. COHEN UNIV OF MASSACHUSETTS COINS DEPT LEDERLE GRC AMHERST MA 01003	1
DR JON DOYLE LABORATORY FOR COMPUTER SCIENCE MASS INSTITUTE OF TECHNOLOGY 545 TECHNOLOGY SQUARE CAMBRIDGE MA 02139	1

DR. BRIAN DRABBLE AI APPLICATIONS INSTITUTE UNIV OF EDINBURGH/80 S. BRIDGE EDINBURGH EH1 LHN UNITED KINGDOM	1
MR. SCOTT FOUSE ISX CORPORATION 4353 PARK TERRACE DRIVE WESTLAKE VILLAGE CA 91361	1
MR. STU DRAPER MITRE EAGLE CENTER 3, SUITE 8 O'FALLON IL 62269	1
DR MARK FOX DEPT OF INDUSTRIAL ENG UNIV OF TORONTO 4 TADDLE CREEK ROAD TORONTO, ONTARIO, CANADA	1
MR. GARY EDWARDS ISX CORPORATION 2000 N 15TH ST, SUITE 1000 ARLINGTON, VA 22201	1
MR. RUSS FREW GENERAL ELECTRIC MOORESTOWN CORPORATE CENTER BLDG ATK 145-2 MOORESTOWN NJ 08057	1
DR MICHAEL FEHLING STANFORD UNIVERSITY ENGINEERING ECO SYSTEMS STANFORD CA 94305	1
RICK HAYES-ROTH CIMFLEX-TEKNOWLEDGE 1810 EMBARCADERO RD PALO ALTO CA 94303	1
DR JIM HENDLER UNIV OF MARYLAND DEPT OF COMPUTER SCIENCE COLLEGE PARK MD 20742	1

MS. YOLANDA GIL USC/ISI 4676 ADMIRALTY WAY MARINA DEL RAY CA 90292	1
MR. MORTON A. HIRSCHBERG, DIRECTOR US ARMY RESEARCH LABORATORY ATTN: AMSRL-CI-CB ABERDEEN PROVING GROUND MD 21005-5066	1
MR. MARK A. HOFFMAN ISX CORPORATION 1165 NORTHCHASE PARKWAY MARIETTA GA 30067	1
DR RON LARSEN NAVAL CMD, CONTROL & OCEAN SUR CTR RESEARCH, DEVELOP, TEST & EVAL DIV CODE 444 SAN DIEGO CA 92152-5000	1
DR CRAIG KNOBLOCK USC-ISI 4676 ADMIRALTY WAY MARINA DEL RAY CA 90292	1
MR. RICHARD LOWE (AP-10) SRA CORPORATION 2000 15TH STREET NORTH ARLINGTON VA 22201	1
MR. TED C. KRAL BBN SYSTEMS & TECHNOLOGIES 4015 HANCOCK STREET, SUITE 101 SAN DIEGO CA 92110	1
DR JOHN LOWRENCE SRI INTERNATIONAL ARTIFICIAL INTELLIGENCE CENTER 333 RAVENSWOOD AVE MENLO PARK CA 94025	1
DR. ALAN MEYROWITZ NAVAL RESEARCH LABORATORY/CODE 5510 4555 OVERLOOK AVE WASH DC 20375	1

ALICE MULVEHILL BBN 10 MOULTON STREET CAMBRIDGE MA 02238	1
DR ROBERT MACGREGOR USC/ISI 4676 ADMIRALTY WAY MARINA DEL REY CA 90292	1
DR DREW MCDERMOTT YALE COMPUTER SCIENCE DEPT P.O. BOX 2153, YALE STATION 51 PROPSPECT STREET MEW HAVEN CT 06520	1
DR DOUGLAS SMITH KESTREL INSTITUTE 3260 HILLVIEW AVE PALO ALTO CA 94304	1
DR. AUSTIN TATE AI APPLICATIONS INSTITUTE UNIV OF EDINBURGH 80 SOUTH BRIDGE EDINBURGH EH1 1HN - SCOTLAND	1
DIRECTOR DARPA/ITO 3701 N. FAIRFAX DR., 7TH FL ARLINGTON VA 22209-1714	1
DR STEPHEN F. SMITH ROBOTICS INSTITUTE/CMU SCHENLEY PRK PITTSBURGH PA 15213	1
DR. ABRAHAM WAKSMAN AFOSR/NM 110 DUNCAN AVE., SUITE B115 BOLLING AFB DC 20331-0001	1
DR JONATHAN P. STILLMAN GENERAL ELECTRIC CRD 1 RIVER RD, RM K1-5C31A P. O. BOX 8 SCHENECTADY NY 12345	1

DR EDWARD C.T. WALKER BBN SYSTEMS & TECHNOLOGIES 10 MOULTON STREET CAMBRIDGE MA 02138	1
DR BILL SWARTOUT USC/ISI 4676 ADMIRALTY WAY MARINA DEL RAY CA 90292	1
GIO WIEDERHOLD STANFORD UNIVERSITY DEPT OF COMPUTER SCIENCE 438 MARGARET JACKS HALL STANFORD CA 94305-2140	1
DR KATIA SYCARA/THE ROBOTICS INST SCHOOL OF COMPUTER SCIENCE CARNEGIE MELLON UNIV DOHERTY HALL RM 3325 PITTSBURGH PA 15213	1
DR DAVID E. WILKINS SRI INTERNATIONAL ARTIFICIAL INTELLIGENCE CENTER 333 RAVENSWOOD AVE MENLO PARK CA 94025	1
DR. PATRICK WINSTON MASS INSTITUTE OF TECHNOLOGY RM NE43-817 545 TECHNOLOGY SQUARE CAMBRIDGE MA 02139	1
DR JOHN P. SCHILL ARPA/ISO 3701 N FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
DR STEVE ROTH CENTER FOR INTEGRATED MANUFACTURING THE ROBOTICS INSTITUTE CARNEGIE MELLON UNIV PITTSBURGH PA 15213-3890	1
DR YOAV SHOHAM STANFORD UNIVERSITY COMPUTER SCIENCE DEPT STANFORD CA 94305	1

MR. MIKE ROUSE AFSC 7800 HAMPTON RD NORFOLK VA 23511-6097	1
DR LARRY BIRNBAUM NORTHWESTERN UNIVERSITY ILS 1890 MAPLE AVE EVANSTON IL 60201	1
MR. LEE ERMAN CIMFLEX TECKNOWLEDGE 1810 EMBARCARDERO RD PALO ALTO CA 94303	1
DR MATTHEW L. GINSBERG CIRL, 1269 UNIVERSITY OF OREGON EUGENE OR 97403	5
MR IRA GOLDSTEIN OPEN SW FOUNDATION RESEARCH INST ONE CAMBRIDGE CENTER CAMBRIDGE MA 02142	1
MR JEFF GROSSMAN, CO NCCOSC RDTE DIV 44 5370 SILVERGATE AVE, ROOM 1405 SAN DIEGO CA 92152-5146	1
JAN GUNTHER ASCENT TECHNOLOGY, INC. 64 SIDNEY ST, SUITE 380 CAMBRIDGE MA 02139	1
DR LYNETTE HIRSCHMAN MITRE CORPORATION 202 BURLINGTON RD BEDFORD MA 01730	1
DR ADELE E. HOWE COMPUTER SCIENCE DEPT COLORADO STATE UNIVERSITY FORT COLLINS CO 80523	1

DR LESLIE PACK KAELBLING COMPUTER SCIENCE DEPT BROWN UNIVERSITY PROVIDENCE RI 02912	1
DR SUBBARAO KAMBHAMPATI DEPT OF COMPUTER SCIENCE ARIZONA STATE UNIVERSITY TEMPE AZ 85287-5406	1
MR THOMAS E. KAZMIERCZAK SRA CORPORATION 331 SALEM PLACE, SUITE 200 FAIRVIEW HEIGHTS IL 62208	1
DR CARLA GOMES ROME LABORATORY/C3CA 525 BROOKS RD ROME NY 13441-4505	1
DR MARK T. MAYBURY ASSOCIATE DIRECTOR OF AI CENTER ADVANCED INFO SYSTEMS TECH G041 MITRE CORP, BURLINGTON RD, MS K-329 BEDFORD MA 01730	1
MR DONALD P. MCKAY PARAMAX/UNISYS P O BOX 517 PAOLI PA 19301	1
DR KAREN MYERS AI CENTER SRI INTERNTIONAL 333 RAVENSWOOD MENLO PARK CA 94025	1
DR MARTHA E POLLACK DEPT OF COMPUTER SCIENCE UNIVERSITY OF PITTSBURGH PITTSBURGH PA 15260	1
DR RAJ REDDY SCHOOL OF COMPUTER SCIENCE CARNEGIE MELLON UNIVERSITY PITTSBURGH PA 15213	1

DR EDWINA RISSLAND 1
DEPT OF COMPUTER & INFO SCIENCE
UNIVERSITY OF MASSACHUSETTS
AMHERST MA 01003

DR MANUELA VELOSO 1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

DR DAN WELD 1
DEPT OF COMPUTER SCIENCE & ENG
MAIL STOP FR-35
UNIVERSITY OF WASHINGTON
SEATTLE WA 98195

MR JOE ROBERTS 1
ISX CORPORATION
4301 N FAIRFAX DRIVE, SUITE 301
ARLINGTON VA 22203

DR TOM GARVEY 1
ARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

MR JOHN N. ENTZMINGER, JR. 1
ARPA/DIRO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

DIRECTOR 1
ARPA/ISO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

OFFICE OF THE CHIEF OF NAVAL RSCH 1
ATTN: MR PAUL QUINN
CODE 311
800 N. QUINCY STREET
ARLINGTON VA 22217

DR OREN ETZIONI 1
DEPT OF COMPUTER SCIENCE
UNIVERSITY OF WASHINGTON
SEATTLE WA 98195

DR GEORGE FERGUSON UNIVERSITY OF ROCHESTER COMPUTER STUDIES BLDG, RM 732 WILSON BLVD ROCHESTER NY 14627	1
DR STEVE HANKS DEPT OF COMPUTER SCIENCE & ENG'G UNIVERSITY OF WASHINGTON SEATTLE WA 98195	1
MR DON MORROW BBN SYSTEMS & TECHOLOGIES 101 MOONGLOW DR BELLEVILLE IL 62221	1
DR CHRISTOPHER OWENS BBN SYSTEMS & TECHNOLOGIES 10 MOULTON ST CAMBRIDGE MA 02138	1
DR ADNAN DARWICHE INFORMATION & DECISION SCIENCES ROCKWELL INT'L SCIENCE CENTER 1049 CAMINO DOS RIOS THOUSAND OAKS CA 91360	1
DR JAIME CARBONNEL THE ROBOTICS INSTITUTE CARNEGIE MELLON UNIVERSITY DOHERTY HALL, ROOM 3325 PITTSBURGH PA 15213	1
DR NORMAN SADEH THE ROBOTICS INSTITUTE CARNEGIE MELLON UNIVERSITY DOHERTY HALL, ROOM 3315 PITTSBURGH PA 15213	1
DR JAMES CRAWFORD CIRL, 1269 UNIVERSITY OF OREGON EUGENE OR 97403	1
DR TAIEB ZNATI UNIVERSITY OF PITTSBURGH DEPT OF COMPUTER SCIENCE PITTSBURGH PA 15260	1

DR MARIE DEJARDINS SRI INTERNATIONAL 333 RAVENSWOOD AVENUE MENLO PARK CA 94025	1
ROBERT J. KRUCHTEN HQ AMC/SCA 203 W LOSEY ST, SUITE 1016 SCOTT AFB IL 62225-5223	1
DR. DAVE GUNNING DARPA/ISO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
MS. LEAH WONG NCCOSC RDTE DIVISIION 53560 HULL STREET SAN DIEGO CA 92152-5001	1
B. MCMURREY NCCOSC RDT&E DIVISION CODE 421 53560 HULL STREET SAN DIEGO CA 95152-5001	1
GINNY ALBERT LOGICON ITG 2100 WASHINGTON BLVD ARLINGTON VA 22204	1
DAVID HESS SAIC ATLANTIC PROGRAMS ONE ENTERPRISE PARKWAY, SUITE 370 HAMPTON VA 23666	1
COL ROBERT PLEBANEK DARPA/ISO 3701 N FAIRFAX DR ARLINGTON VA 22203	1
ADAM PEASE TECKNOWLEDGE 1810 EMBARCADERO RD PALO ALTO CA 94303	1

JIM SHOOP 1
ISX CORPORATION
4353 PARK TERRACE DR
WESTLAKE VILLAGE CA 91361

DR ROBERT NECHES 1
DARPA/ISO
3701 N FAIRFAX DR
ARLINGTON VA 22203

DAVID SWANSON 1
NRAD
53118 GATCHELL RD
44207 BLDG 600 RM 422C
SAN DIEGO CA 92152

DR STEPHEN WESTFOLD 1
KESTREL INSTITUTE
3260 HILLVIEW AVE
PALO ALTO CA 94304

MAJ TOMMY LANCASTER 1
USTRANSCOM/TCJ5-SC
508 SCOTT DR
SCOTT AFB IL 62225-5357

JAMES APPLEGATE 1
MITRE
EAGLE CENTER 3, SUITE 8
O'FALLON IL 62269

SOFTWARE ENGINEERING INSTITUTE 1
CARNEGIE MELLON UNIVERSITY
4500 FIFTH AVE
PITTSBURGH PA 15213

DIRNSA 1
R509
9800 SAVAGE RD
FT MEADE MD 20755-6000

NSA/CSS 1
K1
FT MEADE MD 20755-6000

DCMAO/WICHITA/GKEP SUITE B-34 401 N MARKET ST WICHITA KS 67202-2095	1
PHILLIPS LABORATORY PL/TL (LIBRARY) 5 WRIGHT STREET HANSCOM AFB MA 01731-3004	1
THE MITRE CORPORATION ATTN: E LADURE D460 202 BURLINGTON RD BEDFORD MA 01732	1
OUSD (P)/DTSA/DUTD ATTN: PATRICK G. SULLIVAN, JR 400 ARMY NAVY DR SUITE 300 ARLINGTON VA 22202	1